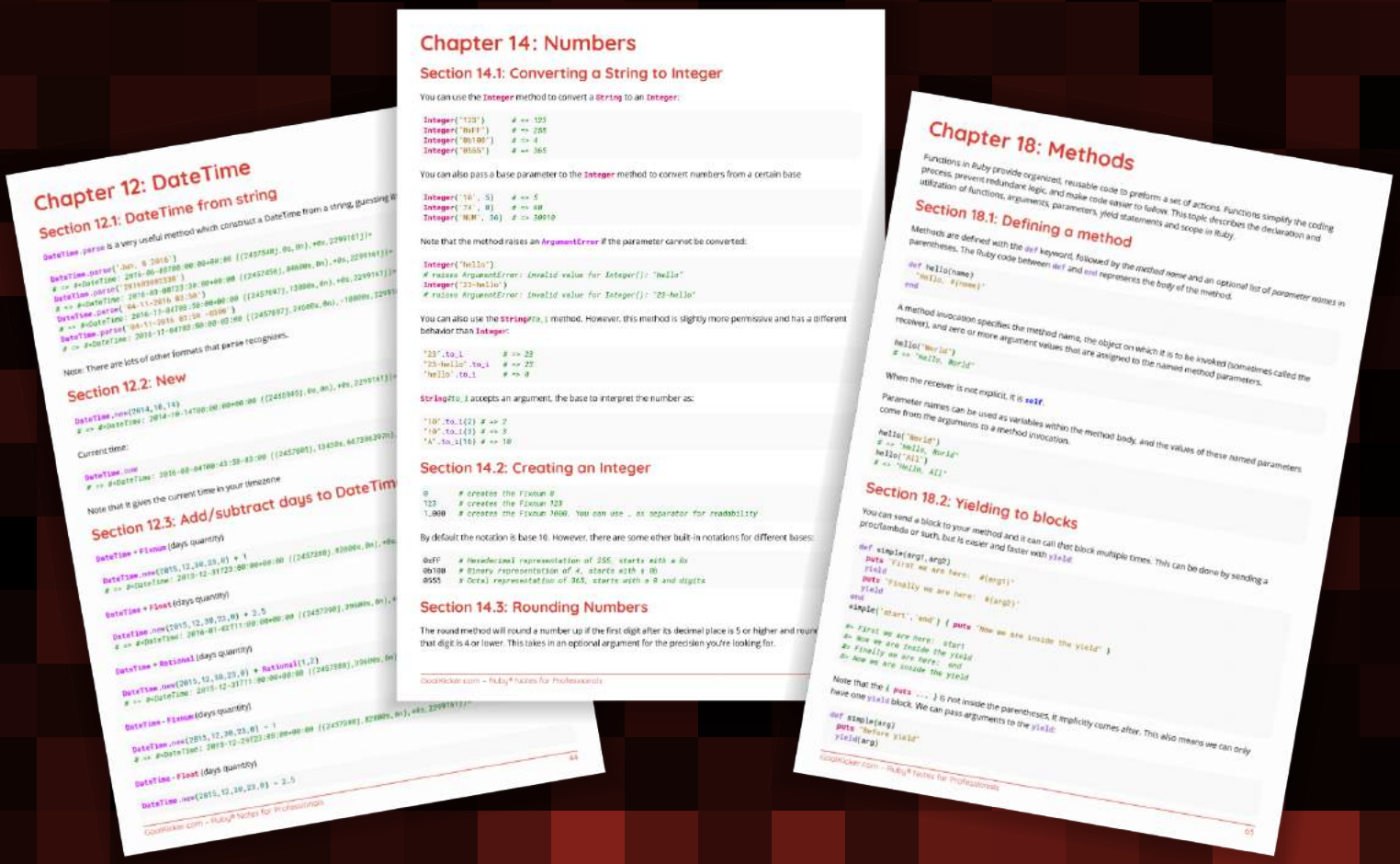
Ruby®

Notes for Professionals



200+ pages

of professional hints and tricks

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
http://GoalKicker.com/RubyBook

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with Ruby Language

Version	Release Date
2.4	2016-12-25
2.3	2015-12-25
2.2	2014-12-25
2.1	2013-12-25
2.0	2013-02-24
1.9	2007-12-25
1.8	2003-08-04
1.6.8	2002-12-24

Section 1.1: Hello World

> This example assumes Ruby is installed.

Place the following in a file named `hello.rb`:

```
puts 'Hello World'
```

From the command line, type the following command to execute the Ruby code from the source file:

```
$ ruby hello.rb
```

This should output:

```
Hello World
```

The output will be immediately displayed to the console. Ruby source files don't need to be compiled before being executed. The Ruby interpreter compiles and executes the Ruby file at runtime.

Section 1.2: Hello World as a Self-Executable File—using Shebang (Unix-like operating systems only)

You can add an interpreter directive (shebang) to your script. Create a file called `hello_world.rb` which contains:

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

puts 'Hello World!'
```

Give the script executable permissions. Here's how to do that in Unix:

```
$ chmod u+x hello_world.rb
```

Now you do not need to call the Ruby interpreter explicitly to run your script.

```
$ ./hello_world.rb
```

Section 1.3: Hello World from IRB

Alternatively, you can use the Interactive Ruby Shell (IRB) to immediately execute the Ruby statements you previously wrote in the Ruby file.

Start an IRB session by typing:

```
$ irb
```

Then enter the following command:

```
puts "Hello World"
```

This results in the following console output (including newline):

```
Hello World
```

If you don't want to start a new line, you can use `print`:

```
print "Hello World"
```

Section 1.4: Hello World without source files

Run the command below in a shell after installing Ruby. This shows how you can execute simple Ruby programs without creating a Ruby file:

```
ruby -e 'puts "Hello World"'
```

You can also feed a Ruby program to the interpreter's standard input. One way to do that is to use a here document in your shell command:

```
ruby <<END
puts "Hello World"
END
```

Section 1.5: Hello World with tk

Tk is the standard graphical user interface (GUI) for Ruby. It provides a cross-platform GUI for Ruby programs.

Example code:

```
require "tk"
TkRoot.new{ title "Hello World!" }
Tk.mainloop
```

The result:

Step by Step explanation:

```
require "tk"
```

Load the tk package.

```
TkRoot.new{ title "Hello World!" }
```

Define a widget with the title `Hello World`

```
Tk.mainloop
```

Start the main loop and display the widget.

Section 1.6: My First Method

Overview

Create a new file named `my_first_method.rb`

Place the following code inside the file:

```
def hello_world
  puts "Hello world!"
end

hello_world() # or just 'hello_world' (without parenthesis)
```

Now, from a command line, execute the following:

```
ruby my_first_method.rb
```

The output should be:

```
Hello world!
```

Explanation

- `def` is a keyword that tells us that we're `def`-ining a method - in this case, `hello_world` is the name of our method.
- `puts "Hello world!"` `puts` (or pipes to the console) the string `Hello world!`
- `end` is a keyword that signifies we're ending our definition of the `hello_world` method

- as the `hello_world` method doesn't accept any arguments, you can omit the parenthesis by invoking the method

Chapter 2: Casting (type conversion)

Section 2.1: Casting to a Float

```
"123.50".to_f   #=> 123.5
Float("123.50") #=> 123.5
```

However, there is a difference when the string is not a valid `Float`:

```
"something".to_f   #=> 0.0
Float("something") # ArgumentError: invalid value for Float(): "something"
```

Section 2.2: Casting to a String

```
123.5.to_s    #=> "123.5"
String(123.5) #=> "123.5"
```

Usually, `String()` will just call `#to_s`.

Methods `Kernel#sprintf` and `String#%` behave similar to C:

```
sprintf("%s", 123.5) #=> "123.5"
"%s" % 123.5 #=> "123.5"
"%d" % 123.5 #=> "123"
"%.2f" % 123.5 #=> "123.50"
```

Section 2.3: Casting to an Integer

```
"123.50".to_i     #=> 123
Integer("123.50") #=> 123
```

A string will take the value of any integer at its start, but will not take integers from anywhere else:

```
"123-foo".to_i # => 123
"foo-123".to_i # => 0
```

However, there is a difference when the string is not a valid Integer:

```
"something".to_i     #=> 0
Integer("something") # ArgumentError: invalid value for Integer(): "something"
```

Section 2.4: Floats and Integers

```
1/2 #=> 0
```

Since we are dividing two integers, the result is an integer. To solve this problem, we need to cast at least one of those to Float:

```
1.0 / 2      #=> 0.5
1.to_f / 2   #=> 0.5
1 / Float(2) #=> 0.5
```

Alternatively, `fdiv` may be used to return the floating point result of division without explicitly casting either

operand:

```
1.fdiv 2 # => 0.5
```

Chapter 3: Operators

Section 3.1: Operator Precedence and Methods

From highest to lowest, this is the precedence table for Ruby. High precedence operations happen before low precedence operations.

Operators	Operations	Method?
.	Method call (e.g. foo.bar)	
[] []=	Bracket Lookup, Bracket Set	✓[1]
! ~ +	Boolean NOT, complement, unary plus	✓[2]
**	Exponentiation	✓
-	Unary minus	✓[2]
* / %	Multiplication, division, modulo	✓
+ -	Addition, subtraction	✓
<>	Bitwise shift	✓
&	Bitwise AND	✓
| ^	Bitwise OR, Bitwise XOR	✓
< <= >= >	Comparison	✓
<=> == != === =~ !~	Equality, pattern matching, comparison	✓[3]
&&	Boolean AND	
||	Boolean OR	
.. ...	Inclusive range, Exclusive range	
? :	Ternary operator	
rescue	Modifier rescue	
= += -=	Assignments	
defined?	Defined operator	
not	Boolean NOT	
or and	Boolean OR, Boolean AND	
if unless while until	Modifier if, unless, while, until	
{ }	Block with braces	
do end	Block with do end	

Unary + and unary - are for `+obj`, `-obj` or `-(some_expression)`.

Modifier-if, modifier-unless, etc. are for the modifier versions of those keywords. For example, this is a modifier-unless expression:

```
a += 1 unless a.zero?
```

Operators with a ✓ may be defined as methods. Most methods are named exactly as the operator is named, for example:

```
class Foo
  def **(x)
    puts "Raising to the power of #{x}"
  end
  def <<(y)
    puts "Shifting left by #{y}"
  end
```

```
  def !
    puts "Boolean negation"
  end
end

Foo.new ** 2      #=> "Raising to the power of 2"
Foo.new << 3      #=> "Shifting left by 3"
!Foo.new          #=> "Boolean negation"
```

[1] The Bracket Lookup and Bracket Set methods ([ ] and []=) have their arguments defined after the name, for example:

```
class Foo
  def [](x)
    puts "Looking up item #{x}"
  end
  def []=(x,y)
    puts "Setting item #{x} to #{y}"
  end
end

f = Foo.new
f[:cats] = 42     #=> "Setting item cats to 42"
f[17]             #=> "Looking up item 17"
```

[2] The "unary plus" and "unary minus" operators are defined as methods named `+@` and `-@`, for example

```
class Foo
  def -@
    puts "unary minus"
  end
  def +@
    puts "unary plus"
  end
end

f = Foo.new
+f                #=> "unary plus"
-f                #=> "unary minus"
```

[3] In early versions of Ruby the inequality operator `!=` and the non-matching operator `!~` could not be defined as methods. Instead, the method for the corresponding equality operator `==` or matching operator `=~` was invoked, and the result of that method was boolean inverted by Ruby.

If you do not define your own `!=` or `!~` operators the above behavior is still true. However, as of Ruby 1.9.1, those two operators may also be defined as methods:

```
class Foo
  def ==(x)
    puts "checking for EQUALITY with #{x}, returning false"
    false
  end
end

f = Foo.new
x = (f == 42)     #=> "checking for EQUALITY with 42, returning false"
puts x            #=> "false"
x = (f != 42)     #=> "checking for EQUALITY with 42, returning false"
puts x            #=> "true"
```

```ruby
class Foo
  def !=(x)
    puts "Checking for INequality with #{x}"
  end
end

f != 42          #=> "checking for INequality with 42"
```

Section 3.2: Case equality operator (===)

Also known as *triple equals*.

This operator does not test equality, but rather tests if the right operand has an IS A relationship with the left operand. As such, the popular name *case equality operator* is misleading.

This SO answer describes it thus: the best way to describe `a === b` is "if I have a drawer labeled `a`, does it make sense to put `b` in it?" In other words, does the set `a` include the member `b`?

Examples (source)

```ruby
(1..5) === 3            # => true
(1..5) === 6            # => false

Integer === 42          # => true
Integer === 'fourtytwo' # => false

/ell/ === 'Hello'       # => true
/ell/ === 'Foobar'      # => false
```

Classes that override `===`

Many classes override `===` to provide meaningful semantics in case statements. Some of them are:

Class	Synonym for
Array	==
Date	==
Module	is_a?
Object	==
Range	include?
Regexp	=~
String	==

Recommended practice

Explicit use of the case equality operator `===` should be avoided. It doesn't test equality but rather *subsumption*, and its use can be confusing. Code is clearer and easier to understand when the synonym method is used instead.

```ruby
# Bad
Integer === 42
(1..5) === 3
/ell/ === 'Hello'

# Good, uses synonym method
42.is_a?(Integer)
(1..5).include?(3)
/ell/ =~ 'Hello'
```

Section 3.3: Safe Navigation Operator

Ruby 2.3.0 added the *safe navigation operator*, `&.`. This operator is intended to shorten the paradigm of `object && object.property && object.property.method` in conditional statements.

For example, you have a `House` object with an `address` property, and you want to find the `street_name` from the `address`. To program this safely to avoid nil errors in older Ruby versions, you'd use code something like this:

```ruby
if house && house.address && house.address.street_name
  house.address.street_name
end
```

The safe navigation operator shortens this condition. Instead, you can write:

```ruby
if house&.address&.street_name
  house.address.street_name
end
```

Caution:
The safe navigation operator doesn't have *exactly* the same behavior as the chained conditional. Using the chained conditional (first example), the `if` block would not be executed if, say `address` was **`false`**. The safe navigation operator only recognises **`nil`** values, but permits values such as **`false`**. If `address` is **`false`**, using the SNO will yield an error:

```ruby
house&.address&.street_name
# => undefined method `address' for false:FalseClass
```

Section 3.4: Assignment Operators

Simple Assignment

`=` is a simple assignment. It creates a new local variable if the variable was not previously referenced.

```ruby
x = 3
y = 4 + 5
puts "x is #{x}, y is #{y}"
```

This will output:

```
x is 3, y is 9
```

Parallel Assignment

Variables can also be assigned in parallel, e.g. `x, y = 3, 9`. This is especially useful for swapping values:

```ruby
x, y = 3, 9
x, y = y, x
puts "x is #{x}, y is #{y}"
```

This will output:

```
x is 9, y is 3
```

Abbreviated Assignment

It's possible to mix operators and assignment. For example:

```ruby
x = 1
y = 2
puts "x is #{x}, y is #{y}"

x += y
puts "x is now #{x}"
```

Shows the following output:

```
x is 1, y is 2
x is now 3
```

Various operations can be used in abbreviated assignment:

Operator	Description	Example	Equivalent to
`+=`	Adds and reassigns the variable	`x += y`	`x = x + y`
`-=`	Subtracts and reassigns the variable	`x -= y`	`x = x - y`
`*=`	Multiplies and reassigns the variable	`x *= y`	`x = x * y`
`/=`	Divides and reassigns the variable	`x /= y`	`x = x / y`
`%=`	Divides, takes the remainder, and reassigns the variable	`x %= y`	`x = x % y`
`**=`	Calculates the exponent and reassigns the variable	`x **= y`	`x = x ** y`

Section 3.5: Comparison Operators

Operator	Description
`==`	`true` if the two values are equal.
`!=`	`true` if the two values are *not* equal.
`<`	`true` if the value of the operand on the left is *less than* the value on the right.
`>`	`true` if the value of the operand on the left is *greater than* the value on the right.
`>=`	`true` if the value of the operand on the left is *greater than* **or** *equal to* the value on the right.
`<=`	`true` if the value of the operand on the left is *less than* **or** *equal to* the value on the right.
`<=>`	`0` if the value of the operand on the left is *equal to* the value on the right, `1` if the value of the operand on the left is *greater than* the value on the right, `-1` if the value of the operand on the left is *less than* the value on the right.

Chapter 4: Variable Scope and Visibility

Section 4.1: Class Variables

Class variables have a class wide scope, they can be declared anywhere in the class. A variable will be considered a class variable when prefixed with `@@`

```
class Dinosaur
    @@classification = "Like a Reptile, but like a bird"

    def self.classification
        @@classification
    end

    def classification
        @@classification
    end
end

dino = Dinosaur.new
dino.classification
# => "Like a Reptile, but like a bird"

Dinosaur.classification
# => "Like a Reptile, but like a bird"
```

Class variables are shared between related classes and can be overwritten from a child class

```
class TRex < Dinosaur
    @@classification = "Big teeth bird!"
end

TRex.classification
# => "Big teeth bird!"

Dinosaur.classification
# => "Big teeth bird!"
```

This behaviour is unwanted most of the time and can be circumvented by using class-level instance variables.

Class variables defined inside a module will not overwrite their including classes class variables:

```
module SomethingStrange
    @@classification = "Something Strange"
end

class DuckDinosaur < Dinosaur
    include SomethingStrange
end

DuckDinosaur.class_variables
# => [:@@classification]
SomethingStrange.class_variables
# => [:@@classification]

DuckDinosaur.classification
# => "Big teeth bird!"
```

Section 4.2: Local Variables

Local variables (unlike the other variable classes) do not have any prefix

```
local_variable = "local"
p local_variable
# => local
```

Its scope is dependent on where it has been declared, it can not be used outside the "declaration containers" scope. For example, if a local variable is declared in a method, it can only be used inside that method.

```
def some_method
    method_scope_var = "hi there"
    p method_scope_var
end

some_method
# hi there
# => hi there

method_scope_var
# NameError: undefined local variable or method `method_scope_var'
```

Of course, local variables are not limited to methods, as a rule of thumb you could say that, as soon as you declare a variable inside a `do ... end` block or wrapped in curly braces {} it will be local and scoped to the block it has been declared in.

```
2.times do |n|
    local_var = n + 1
    p local_var
end
# 1
# 2
# => 2

local_var
# NameError: undefined local variable or method `local_var'
```

However, local variables declared in `if` or `case` blocks can be used in the parent-scope:

```
if true
    usable = "yay"
end

p usable
# yay
# => "yay"
```

While local variables can not be used outside of its block of declaration, it will be passed down to blocks:

```
my_variable = "foo"

my_variable.split("").each_with_index do |char, i|
    puts "The character in string '#{my_variable}' at index #{i} is #{char}"
end
# The character in string 'foo' at index 0 is f
# The character in string 'foo' at index 1 is o
# The character in string 'foo' at index 2 is o
```

```ruby
# => ["f", "o", "o"]
```

But not to method / class / module definitions

```ruby
my_variable = "foo"

def some_method
    puts "you can't use the local variable in here, see? #{my_variable}"
end

some_method
# NameError: undefined local variable or method `my_variable'
```

The variables used for block arguments are (of course) local to the block, but will overshadow previously defined variables, without overwriting them.

```ruby
overshadowed = "sunlight"

["darkness"].each do |overshadowed|
    p overshadowed
end
# darkness
# => ["darkness"]

p overshadowed
# "sunlight"
# => "sunlight"
```

Section 4.3: Global Variables

Global variables have a global scope and hence, can be used everywhere. Their scope is not dependent on where they are defined. A variable will be considered global, when prefixed with a $ sign.

```ruby
$i_am_global = "omg"

class Dinosaur
    def instance_method
       p "global vars can be used everywhere. See? #{$i_am_global}, #{$another_global_var}"
    end

    def self.class_method
       $another_global_var = "srsly?"
       p "global vars can be used everywhere. See? #{$i_am_global}"
    end
end

Dinosaur.class_method
# "global vars can be used everywhere. See? omg"
# => "global vars can be used everywhere. See? omg"

dinosaur = Dinosaur.new
dinosaur.instance_method
# "global vars can be used everywhere. See? omg, srsly?"
# => "global vars can be used everywhere. See? omg, srsly?"
```

Since a global variable can be defined everywhere and will be visible everywhere, calling an "undefined" global variable will return nil instead of raising an error.

```ruby
p $undefined_var
# nil
# => nil
```

Although global variables are easy to use its usage is strongly discouraged in favour of constants.

Section 4.4: Instance Variables

Instance variables have an object wide scope, they can be declared anywhere in the object, however an instance variable declared on class level, will only be visible in the class object. A variable will be considered an instance variable when prefixed with `@`. Instance variables are used to set and get an objects attributes and will return nil if not defined.

```ruby
class Dinosaur
    @base_sound = "rawrr"

    def initialize(sound = nil)
        @sound = sound || self.class.base_sound
    end

    def speak
        @sound
    end

    def try_to_speak
        @base_sound
    end

    def count_and_store_sound_length
        @sound.chars.each_with_index do |char, i|
            @sound_length = i + 1
            p "#{char}: #{sound_length}"
        end
    end

    def sound_length
        @sound_length
    end

    def self.base_sound
        @base_sound
    end
end

dino_1 = Dinosaur.new
dino_2 = Dinosaur.new "grrr"

Dinosaur.base_sound
# => "rawrr"
dino_2.speak
# => "grrr"
```

The instance variable declared on class level can not be accessed on object level:

```ruby
dino_1.try_to_speak
# => nil
```

However, we used the instance variable `@base_sound` to instantiate the sound when no sound is passed to the new

method:

```
dino_1.speak
# => "rawwr"
```

Instance variables can be declared anywhere in the object, even inside a block:

```
dino_1.count_and_store_sound_length
# "r: 1"
# "a: 2"
# "w: 3"
# "r: 4"
# "r: 5"
# => ["r", "a", "w", "r", "r"]

dino_1.sound_length
# => 5
```

Instance variables are **not** shared between instances of the same class

```
dino_2.sound_length
# => nil
```

This can be used to create class level variables, that will not be overwritten by a child-class, since classes are also objects in Ruby.

```
class DuckDuckDinosaur < Dinosaur
    @base_sound = "quack quack"
end

duck_dino = DuckDuckDinosaur.new
duck_dino.speak
# => "quack quack"
DuckDuckDinosaur.base_sound
# => "quack quack"
Dinosaur.base_sound
# => "rawrr"
```

Chapter 5: Environment Variables

Section 5.1: Sample to get user profile path

```
# will retrieve my home path
ENV['HOME'] # => "/Users/username"

# will try retrieve the 'FOO' environment variable. If failed, will get 'bar'
ENV.fetch('FOO', 'bar')
```

Chapter 6: Constants

Section 6.1: Define a constant

```
MY_CONSTANT = "Hello, world" # constant
Constant = 'This is also constant' # constant
my_variable = "Hello, venus" # not constatn
```

Constant name start with capital letter. Everything that start with capital letter are considered as `constant` in Ruby. So `class` and `module` are also constant. Best practice is use all capital letter for declaring constant.

Section 6.2: Modify a Constant

```
MY_CONSTANT = "Hello, world"
MY_CONSTANT = "Hullo, world"
```

The above code results in a warning, because you should be using variables if you want to change their values. However it is possible to change one letter at a time in a constant without a warning, like this:

```
MY_CONSTANT = "Hello, world"
MY_CONSTANT[1] = "u"
```

Now, after changing the second letter of `MY_CONSTANT`, it becomes `"Hullo, world"`.

Section 6.3: Constants cannot be defined in methods

```
def say_hi
  MESSAGE = "Hello"
  puts MESSAGE
end
```

The above code results in an error: `SyntaxError: (irb):2: dynamic constant assignment`.

Section 6.4: Define and change constants in a class

```
class Message
  DEFAULT_MESSAGE = "Hello, world"

  def speak(message = nil)
    if message
      puts message
    else
      puts DEFAULT_MESSAGE
    end
  end
end
```

The constant `DEFAULT_MESSAGE` can be changed with the following code:

```
Message::DEFAULT_MESSAGE = "Hullo, world"
```

Chapter 7: Special Constants in Ruby

Section 7.1: __FILE__

Is the relative path to the file from the current execution directory
Assume we have this directory structure: /home/stackoverflow/script.rb
script.rb contains:

```
puts __FILE__
```

If you are inside /home/stackoverflow and execute the script like `ruby script.rb` then `__FILE__` will output `script.rb` If you are inside /home then it will output `stackoverflow/script.rb`

Very useful to get the path of the script in versions prior to 2.0 where `__dir__` doesn't exist.

Note `__FILE__` is not equal to `__dir__`

Section 7.2: __dir__

`__dir__` is not a constant but a function
`__dir__` is equal to `File.dirname(File.realpath(__FILE__))`

Section 7.3: $PROGRAM_NAME or $0

Contains the name of the script being executed.
Is the same as `__FILE__` if you are executing that script.

Section 7.4: $$

The process number of the Ruby running this script

Section 7.5: $1, $2, etc

Contains the subpattern from the corresponding set of parentheses in the last successful pattern matched, not counting patterns matched in nested blocks that have been exited already, or nil if the last pattern match failed. These variables are all read-only.

Section 7.6: ARGV or $*

Command line arguments given for the script. The options for Ruby interpreter are already removed.

Section 7.7: STDIN

The standard input. The default value for $stdin

Section 7.8: STDOUT

The standard output. The default value for $stdout

Section 7.9: STDERR

The standard error output. The default value for $stderr

Section 7.10: $stderr

The current standard error output.

Section 7.11: $stdout

The current standard output

Section 7.12: $stdin

The current standard input

Section 7.13: ENV

The hash-like object contains current environment variables. Setting a value in ENV changes the environment for child processes.

Chapter 8: Comments

Section 8.1: Single & Multiple line comments

Comments are programmer-readable annotations that are ignored at runtime. Their purpose is to make source code easier to understand.

Single line comments

The `#` character is used to add single line comments.

```
#!/usr/bin/ruby -w
# This is a single line comment.
puts "Hello World!"
```

When executed, the above program will output `Hello World!`

Multiline comments

Multiple-line comments can be added by using `=begin` and `=end` syntax (also known as the comment block markers) as follows:

```
#!/usr/bin/ruby -w
=begin
This is a multiline comment.
Write as many line as you want.
=end
puts "Hello World!"
```

When executed, the above program will output `Hello World!`

Chapter 9: Arrays

Section 9.1: Create Array of Strings

Arrays of strings can be created using ruby's percent string syntax:

```
array = %w(one two three four)
```

This is functionally equivalent to defining the array as:

```
array = ['one', 'two', 'three', 'four']
```

Instead of `%w()` you may use other matching pairs of delimiters: `%w{...}`, `%w[...]` or `%w<...>`.

It is also possible to use arbitrary non-alphanumeric delimiters, such as: `%w!...!`, `%w#...#` or `%w@...@`.

`%W` can be used instead of `%w` to incorporate string interpolation. Consider the following:

```
var = 'hello'

%w(#{var}) # => ["\#{var}"]
%W(#{var}) # => ["hello"]
```

Multiple words can be interpreted by escaping the space with a \.

```
%w(Colorado California New\ York) # => ["Colorado", "California", "New York"]
```

Section 9.2: Create Array with Array::new

An empty Array ([ ]) can be created with Array's class method, `Array::new`:

```
Array.new
```

To set the length of the array, pass a numerical argument:

```
Array.new 3 #=> [nil, nil, nil]
```

There are two ways to populate an array with default values:

- Pass an immutable value as second argument.
- Pass a block that gets current index and generates mutable values.

```
Array.new 3, :x #=> [:x, :x, :x]

Array.new(3) { |i| i.to_s } #=> ["0", "1", "2"]

a = Array.new 3, "X"            # Not recommended.
a[1].replace "C"                # a => ["C", "C", "C"]

b = Array.new(3) { "X" }        # The recommended way.
b[1].replace "C"                # b => ["X", "C", "X"]
```

Section 9.3: Create Array of Symbols

Version ≥ 2.0

```
array = %i(one two three four)
```

Creates the array `[:one, :two, :three, :four]`.

Instead of `%i(...)`, you may use `%i{...}` or `%i[...]` or `%i!...!`

Additionally, if you want to use interpolation, you can do this with `%I`.

Version ≥ 2.0

```
a = 'hello'
b = 'goodbye'
array_one = %I(#{a} #{b} world)
array_two = %i(#{a} #{b} world)
```

Creates the arrays: `array_one = [:hello, :goodbye, :world]` and `array_two = [:"\#{a}", :"\#{b}", :world]`

Section 9.4: Manipulating Array Elements

Adding elements:

```
[1, 2, 3] << 4
# => [1, 2, 3, 4]

[1, 2, 3].push(4)
# => [1, 2, 3, 4]

[1, 2, 3].unshift(4)
# => [4, 1, 2, 3]

[1, 2, 3] << [4, 5]
# => [1, 2, 3, [4, 5]]
```

Removing elements:

```
array = [1, 2, 3, 4]
array.pop
# => 4
array
# => [1, 2, 3]

array = [1, 2, 3, 4]
array.shift
# => 1
array
# => [2, 3, 4]

array = [1, 2, 3, 4]
array.delete(1)
# => 1
array
# => [2, 3, 4]

array = [1,2,3,4,5,6]
array.delete_at(2) // delete from index 2
# => 3
array
```

```ruby
# => [1,2,4,5,6]


array = [1, 2, 2, 2, 3]
array - [2]
# => [1, 3]    # removed all the 2s
array - [2, 3, 4]
# => [1]       # the 4 did nothing
```

Combining arrays:

```ruby
[1, 2, 3] + [4, 5, 6]
# => [1, 2, 3, 4, 5, 6]

[1, 2, 3].concat([4, 5, 6])
# => [1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6] - [2, 3]
# => [1, 4, 5, 6]

[1, 2, 3] | [2, 3, 4]
# => [1, 2, 3, 4]

[1, 2, 3] & [3, 4]
# => [3]
```

You can also multiply arrays, e.g.

```ruby
[1, 2, 3] * 2
# => [1, 2, 3, 1, 2, 3]
```

Section 9.5: Accessing elements

You can access the elements of an array by their indices. Array index numbering starts at `0`.

```ruby
%w(a b c)[0] # => 'a'
%w(a b c)[1] # => 'b'
```

You can crop an array using range

```ruby
%w(a b c d)[1..2] # => ['b', 'c'] (indices from 1 to 2, including the 2)
%w(a b c d)[1...2] # => ['b'] (indices from 1 to 2, excluding the 2)
```

This returns a new array, but doesn't affect the original. Ruby also supports the use of negative indices.

```ruby
%w(a b c)[-1] # => 'c'
%w(a b c)[-2] # => 'b'
```

You can combine negative and positive indices as well

```ruby
%w(a b c d e)[1...-1] # => ['b', 'c', 'd']
```

Other useful methods

Use `first` to access the first element in an array:

```ruby
[1, 2, 3, 4].first # => 1
```

Or `first(n)` to access the first `n` elements returned in an array:

```
[1, 2, 3, 4].first(2) # => [1, 2]
```

Similarly for `last` and `last(n)`:

```
[1, 2, 3, 4].last    # => 4
[1, 2, 3, 4].last(2) # => [3, 4]
```

Use `sample` to access a random element in a array:

```
[1, 2, 3, 4].sample  # => 3
[1, 2, 3, 4].sample  # => 1
```

Or `sample(n)`:

```
[1, 2, 3, 4].sample(2) # => [2, 1]
[1, 2, 3, 4].sample(2) # => [3, 4]
```

Section 9.6: Creating an Array with the literal constructor [ ]

Arrays can be created by enclosing a list of elements in square brackets (`[` and `]`). Array elements in this notation are separated with commas:

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

Arrays can contain any kind of objects in any combination with no restrictions on type:

```
array = [1, 'b', nil, [3, 4]]
```

Section 9.7: Decomposition

Any array can be quickly **decomposed** by assigning its elements into multiple variables. A simple example:

```
arr = [1, 2, 3]
# ---
a = arr[0]
b = arr[1]
c = arr[2]
# --- or, the same
a, b, c = arr
```

Preceding a variable with the *splat* operator (`*`) puts into it an array of all the elements that haven't been captured by other variables. If none are left, empty array is assigned. Only one splat can be used in a single assignment:

```
a, *b = arr       # a = 1; b = [2, 3]
a, *b, c = arr    # a = 1; b = [2]; c = 3
a, b, c, *d = arr # a = 1; b = 2; c = 3; d = []
a, *b, *c = arr   # SyntaxError: unexpected *
```

Decomposition is *safe* and never raises errors. `nil`s are assigned where there's not enough elements, matching the behavior of `[ ]` operator when accessing an index out of bounds:

```
arr[9000] # => nil
```

```ruby
a, b, c, d = arr # a = 1; b = 2; c = 3; d = nil
```

Decomposition tries to call `to_ary` implicitly on the object being assigned. By implementing this method in your type you get the ability to decompose it:

```ruby
class Foo
  def to_ary
    [1, 2]
  end
end
a, b = Foo.new # a = 1; b = 2
```

If the object being decomposed doesn't `respond_to? to_ary`, it's treated as a single-element array:

```ruby
1.respond_to?(:to_ary) # => false
a, b = 1 # a = 1; b = nil
```

Decomposition can also be **nested** by using a `()`-delimited decomposition expression in place of what otherwise would be a single element:

```ruby
arr = [1, [2, 3, 4], 5, 6]
a, (b, *c), *d = arr # a = 1; b = 2; c = [3, 4]; d = [5, 6]
#   ^^^^^
```

This is effectively the opposite of *splat*.

Actually, any decomposition expression can be delimited by `()`. But for the first level decomposition is optional.

```ruby
a, b = [1, 2]
(a, b) = [1, 2] # the same thing
```

Edge case: *a single identifier* cannot be used as a destructuring pattern, be it outer or a nested one:

```ruby
(a) = [1] # SyntaxError
a, (b) = [1, [2]] # SyntaxError
```

When assigning **an array literal** to a destructuring expression, outer `[]` can be omitted:

```ruby
a, b = [1, 2]
a, b =  1, 2  # exactly the same
```

This is known as **parallel assignment**, but it uses the same decomposition under the hood. This is particularly handy for exchanging variables' values without employing additional temporary variables:

```ruby
t = a; a = b; b = t # an obvious way
a, b = b, a         # an idiomatic way
(a, b) = [b, a]     # ...and how it works
```

Values are captured when building the right-hand side of the assignment, so using the same variables as source and destination is relatively safe.

Section 9.8: Arrays union, intersection and difference

```ruby
x = [5, 5, 1, 3]
y = [5, 2, 4, 3]
```

Union (`|`) contains elements from both arrays, with duplicates removed:

```
x | y
=> [5, 1, 3, 2, 4]
```

Intersection (`&`) contains elements which are present both in first and second array:

```
x & y
=> [5, 3]
```

Difference (-) contains elements which are present in first array and not present in second array:

```
x - y
=> [1]
```

Section 9.9: Remove all nil elements from an array with #compact

If an array happens to have one or more `nil` elements and these need to be removed, the `Array#compact` or `Array#compact!` methods can be used, as below.

```
array = [ 1, nil, 'hello', nil, '5', 33]

array.compact # => [ 1, 'hello', '5', 33]

#notice that the method returns a new copy of the array with nil removed,
#without affecting the original

array = [ 1, nil, 'hello', nil, '5', 33]

#If you need the original array modified, you can either reassign it

array = array.compact # => [ 1, 'hello', '5', 33]

array = [ 1, 'hello', '5', 33]

#Or you can use the much more elegant 'bang' version of the method

array = [ 1, nil, 'hello', nil, '5', 33]

array.compact # => [ 1, 'hello', '5', 33]

array = [ 1, 'hello', '5', 33]
```

Finally, notice that if `#compact` or `#compact!` are called on an array with no `nil` elements, these will return nil.

```
array = [ 'foo', 4, 'life']

array.compact # => nil

array.compact! # => nil
```

Section 9.10: Get all combinations / permutations of an array

The `permutation` method, when called with a block yields a two dimensional array consisting of all ordered sequences of a collection of numbers.

If this method is called without a block, it will return an `enumerator`. To convert to an array, call the `to_a` method.

Example	Result
`[1,2,3].permutation`	`#<Enumerator: [1,2,3]:permutation`
`[1,2,3].permutation.to_a`	`[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]`
`[1,2,3].permutation(2).to_a`	`[[1,2],[1,3],[2,1],[2,3],[3,1],[3,2]]`
`[1,2,3].permutation(4).to_a`	[ ] -> No permutations of length 4

The `combination` method on the other hand, when called with a block yields a two-dimensional array consisting of all sequences of a collection of numbers. Unlike permutation, order is disregarded in combinations. For example, `[1,2,3]` is the same as `[3,2,1]`

Example	Result
`[1,2,3].combination(1)`	`#<Enumerator: [1,2,3]:combination`
`[1,2,3].combination(1).to_a`	`[[1],[2],[3]]`
`[1,2,3].combination(3).to_a`	`[[1,2,3]]`
`[1,2,3].combination(4).to_a`	[ ] -> No combinations of length 4

Calling the combination method by itself will result in an enumerator. To get an array, call the `to_a` method.

The `repeated_combination` and `repeated_permutation` methods are similar, except the same element can be repeated multiple times.

For example the sequences `[1,1]`, `[1,3,3,1]`, `[3,3,3]` would not be valid in regular combinations and permutations.

Example	# Combos
`[1,2,3].combination(3).to_a.length`	1
`[1,2,3].repeated_combination(3).to_a.length`	6
`[1,2,3,4,5].combination(5).to_a.length`	1
`[1,2,3].repeated_combination(5).to_a.length`	126

Section 9.11: Inject, reduce

Inject and reduce are different names for the same thing. In other languages these functions are often called folds (like foldl or foldr). These methods are available on every Enumerable object.

Inject takes a two argument function and applies that to all of the pairs of elements in the Array.

For the array `[1, 2, 3]` we can add all of these together with the starting value of zero by specifying a starting value and block like so:

```
[1,2,3].reduce(0) {|a,b| a + b} # => 6
```

Here we pass the function a starting value and a block that says to add all of the values together. The block is first run with `0` as `a` and `1` as `b` it then takes the result of that as the next `a` so we are then adding `1` to the second value `2`. Then we take the result of that (3) and add that on to the final element in the list (also 3) giving us our result (6).

If we omit the first argument, it will set `a` to being the first element in the list, so the example above is the same as:

```
[1,2,3].reduce {|a,b| a + b} # => 6
```

In addition, instead of passing a block with a function, we can pass a named function as a symbol, either with a starting value, or without. With this, the above example could be written as:

```ruby
[1,2,3].reduce(0, :+) # => 6
```

or omitting the starting value:

```ruby
[1,2,3].reduce(:+) # => 6
```

Section 9.12: Filtering arrays

Often we want to operate only on elements of an array that fulfill a specific condition:

Select

Will return elements that match a specific condition

```ruby
array = [1, 2, 3, 4, 5, 6]
array.select { |number| number > 3 } # => [4, 5, 6]
```

Reject

Will return elements that do not match a specific condition

```ruby
array = [1, 2, 3, 4, 5, 6]
array.reject { |number| number > 3 } # => [1, 2, 3]
```

Both `#select` and `#reject` return an array, so they can be chained:

```ruby
array = [1, 2, 3, 4, 5, 6]
array.select { |number| number > 3 }.reject { |number| number < 5 }
  # => [5, 6]
```

Section 9.13: #map

`#map`, provided by Enumerable, creates an array by invoking a block on each element and collecting the results:

```ruby
[1, 2, 3].map { |i| i * 3 }
# => [3, 6, 9]

['1', '2', '3', '4', '5'].map { |i| i.to_i }
# => [1, 2, 3, 4, 5]
```

The original array is not modified; a new array is returned containing the transformed values in the same order as the source values. `map!` can be used if you want to modify the original array.

In `map` method you can call method or use proc to all elements in array.

```ruby
# call to_i method on all elements
%w(1 2 3 4 5 6 7 8 9 10).map(&:to_i)
# => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# using proc (lambda) on all elements
%w(1 2 3 4 5 6 7 8 9 10).map(&->(i){ i.to_i * 2})
# => [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

`map` is synonymous with `collect`.

Section 9.14: Arrays and the splat (*) operator

The `*` operator can be used to unpack variables and arrays so that they can be passed as individual arguments to a method.

This can be used to wrap a single object in an Array if it is not already:

```
def wrap_in_array(value)
  [*value]
end

wrap_in_array(1)
#> [1]

wrap_in_array([1, 2, 3])
#> [1, 2, 3]

wrap_in_array(nil)
#> []
```

In the above example, the `wrap_in_array` method accepts one argument, `value`.

If `value` is an `Array`, its elements are unpacked and a new array is created containing those element.

If `value` is a single object, a new array is created containing that single object.

If `value` is `nil`, an empty array is returned.

The splat operator is particularly handy when used as an argument in methods in some cases. For example, it allows `nil`, single values and arrays to be handled in a consistent manner:

```
def list(*values)
  values.each do |value|
    # do something with value
    puts value
  end
end

list(100)
#> 100

list([100, 200])
#> 100
#> 200

list(nil)
# nothing is outputted
```

Section 9.15: Two-dimensional array

Using the `Array::new` constructor, your can initialize an array with a given size and a new array in each of its slots. The inner arrays can also be given a size and and initial value.

For instance, to create a 3x4 array of zeros:

```
array = Array.new(3) { Array.new(4) { 0 } }
```

The array generated above looks like this when printed with `p`:

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

You can read or write to elements like this:

```
x = array[0][1]
array[2][3] = 2
```

Section 9.16: Turn multi-dimensional array into a one-dimensional (flattened) array

```
[1, 2, [[3, 4], [5]], 6].flatten  # => [1, 2, 3, 4, 5, 6]
```

If you have a multi-dimensional array and you need to make it a *simple* (i.e. one-dimensional) array, you can use the ***#flatten*** method.

Section 9.17: Get unique array elements

In case you need to read an array elements ***avoiding repetitions*** you case use the ***#uniq*** method:

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq
#=> [1, 2, 3, 4, 5]
```

Instead, if you want to remove all duplicated elements from an array, you may use ***#uniq!*** method:

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq!
#=> [1, 2, 3, 4, 5]
```

While the output is the same, ***#uniq!*** also stores the new array:

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq
#=> [1, 2, 3, 4, 5]
a
#=> [1, 1, 2, 3, 4, 4, 5]

a = [1, 1, 2, 3, 4, 4, 5]
a.uniq!
#=> [1, 2, 3, 4, 5]
a
#=> [1, 2, 3, 4, 5]
```

Section 9.18: Create Array of numbers

The normal way to create an array of numbers:

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

Range objects can be used extensively to create an array of numbers:

```
numbers = Array(1..10) # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```ruby
numbers = (1..10).to_a # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`#step` and `#map` methods allow us to impose conditions on the range of numbers:

```ruby
odd_numbers = (1..10).step(2).to_a # => [1, 3, 5, 7, 9]

even_numbers = 2.step(10, 2).to_a # => [2, 4, 6, 8, 10]

squared_numbers = (1..10).map { |number| number * number } # => [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

All the above methods load the numbers eagerly. If you have to load them lazily:

```ruby
number_generator = (1..100).lazy # => #<Enumerator::Lazy: 1..100>

number_generator.first(10) # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Section 9.19: Create an Array of consecutive numbers or letters

This can be easily accomplished by calling `Enumerable#to_a` on a `Range` object:

```ruby
(1..10).to_a    #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`(a..b)` means that it will include all numbers between a and b. To exclude the last number, use `a...b`

```ruby
a_range = 1...5
a_range.to_a       #=> [1, 2, 3, 4]
```

or

```ruby
('a'..'f').to_a    #=> ["a", "b", "c", "d", "e", "f"]
('a'...'f').to_a   #=> ["a", "b", "c", "d", "e"]
```

A convenient shortcut for creating an array is `[*a..b]`

```ruby
[*1..10]           #=> [1,2,3,4,5,6,7,8,9,10]
[*'a'..'f']        #=> ["a", "b", "c", "d", "e", "f"]
```

Section 9.20: Cast to Array from any object

To get Array from any object, use `Kernel#Array`.

The following is an example:

```ruby
Array('something') #=> ["something"]
Array([2, 1, 5])   #=> [2, 1, 5]
Array(1)           #=> [1]
Array(2..4)        #=> [2, 3, 4]
Array([])          #=> []
Array(nil)         #=> []
```

For example, you could replace `join_as_string` method from the following code

```ruby
def join_as_string(arg)
```

```
  if arg.instance_of?(Array)
    arg.join(',')
  elsif arg.instance_of?(Range)
    arg.to_a.join(',')
  else
    arg.to_s
  end
end

join_as_string('something') #=> "something"
join_as_string([2, 1, 5])   #=> "2,1,5"
join_as_string(1)           #=> "1"
join_as_string(2..4)        #=> "2,3,4"
join_as_string([])          #=> ""
join_as_string(nil)         #=> ""
```

to the following code.

```
def join_as_string(arg)
  Array(arg).join(',')
end
```

Chapter 10: Multidimensional Arrays

Multidimensional Arrays in Ruby are just arrays whose elements are other arrays.

The only catch is that since Ruby arrays can contain elements of mixed types, you must be confident that the array that you are manipulating is effectively composed of other arrays and not, for example, arrays and strings.

Section 10.1: Initializing a 2D array

Let's first recap how to initialize a 1D ruby array of integers:

```
my_array = [1, 1, 2, 3, 5, 8, 13]
```

Being a 2D array simply an array of arrays, you can initialize it like this:

```
my_array = [
  [1, 1, 2, 3, 5, 8, 13],
  [1, 4, 9, 16, 25, 36, 49, 64, 81],
  [2, 3, 5, 7, 11, 13, 17]
]
```

Section 10.2: Initializing a 3D array

You can go a level further down and add a third layer of arrays. The rules don't change:

```
my_array = [
  [
    [1, 1, 2, 3, 5, 8, 13],
    [1, 4, 9, 16, 25, 36, 49, 64, 81],
    [2, 3, 5, 7, 11, 13, 17]
  ],
  [
    ['a', 'b', 'c', 'd', 'e'],
    ['z', 'y', 'x', 'w', 'v']
  ],
  [
    []
  ]
]
```

Section 10.3: Accessing a nested array

Accessing the 3rd element of the first subarray:

```
my_array[1][2]
```

Section 10.4: Array flattening

Given a multidimensional array:

```
my_array = [[1, 2], ['a', 'b']]
```

the operation of flattening is to decompose all array children into the root array:

```ruby
my_array.flatten

# [1, 2, 'a', 'b']
```

Chapter 11: Strings

Section 11.1: Difference between single-quoted and double-quoted String literals

The main difference is that double-quoted `String` literals support string interpolations and the full set of escape sequences.

For instance, they can include arbitrary Ruby expressions via interpolation:

```
# Single-quoted strings don't support interpolation
puts 'Now is #{Time.now}'
# Now is #{Time.now}

# Double-quoted strings support interpolation
puts "Now is #{Time.now}"
# Now is 2016-07-21 12:43:04 +0200
```

Double-quoted strings also support the entire set of escape sequences including `"\n"`, `"\t"`...

```
puts 'Hello\nWorld'
# Hello\nWorld

puts "Hello\nWorld"
# Hello
# World
```

... while single-quoted strings support *no* escape sequences, baring the minimal set necessary for single-quoted strings to be useful: Literal single quotes and backslashes, `'\''` and `'\\'` respectively.

Section 11.2: Creating a String

Ruby provides several ways to create a `String` object. The most common way is using single or double quotes to create a "string literal":

```
s1 = 'Hello'
s2 = "Hello"
```

The main difference is that double-quoted string literals are a little bit more flexible as they support interpolation and some backslash escape sequences.

There are also several other possible ways to create a string literal using arbitrary string delimiters. An arbitrary string delimiter is a `%` followed by a matching pair of delimiters:

```
%(A string)
%{A string}
%<A string>
%|A string|
%!A string!
```

Finally, you can use the `%q` and `%Q` sequence, that are equivalent to `'` and `"`:

```
puts %q(A string)
# A string
puts %q(Now is #{Time.now})
```

```ruby
# Now is #{Time.now}

puts %Q(A string)
# A string
puts %Q(Now is #{Time.now})
# Now is 2016-07-21 12:47:45 +0200
```

`%q` and `%Q` sequences are useful when the string contains either single quotes, double quotes, or a mix of both. In this way, you don't need to escape the content:

```ruby
%Q(<a href="/profile">User's profile<a>)
```

You can use several different delimiters, as long as there is a matching pair:

```ruby
%q(A string)
%q{A string}
%q<A string>
%q|A string|
%q!A string!
```

Section 11.3: Case manipulation

```ruby
"string".upcase     # => "STRING"
"STRING".downcase   # => "string"
"String".swapcase   # => "sTRING"
"string".capitalize # => "String"
```

These four methods do not modify the original receiver. For example,

```ruby
str = "Hello"
str.upcase # => "HELLO"
puts str   # => "Hello"
```

There are four similar methods that perform the same actions but modify original receiver.

```ruby
"string".upcase!     # => "STRING"
"STRING".downcase!   # => "string"
"String".swapcase!   # => "sTRING"
"string".capitalize! # => "String"
```

For example,

```ruby
str = "Hello"
str.upcase!  # => "HELLO"
puts str     # => "HELLO"
```

Notes:

- prior to Ruby 2.4 these methods do not handle unicode.

Section 11.4: String concatenation

Concatenate strings with the `+` operator:

```ruby
s1 = "Hello"
s2 = " "
```

```ruby
s3 = "World"

puts s1 + s2 + s3
# => Hello World

s = s1 + s2 + s3
puts s
# => Hello World
```

Or with the `<<` operator:

```ruby
s = 'Hello'
s << ' '
s << 'World'
puts s
# => Hello World
```

Note that the `<<` operator modifies the object on the left hand side.

You also can multiply strings, e.g.

```ruby
"wow" * 3
# => "wowwowwow"
```

Section 11.5: Positioning strings

In Ruby, strings can be left-justified, right-justified or centered

To left-justify string, use the `ljust` method. This takes in two parameters, an integer representing the number of characters of the new string and a string, representing the pattern to be filled.

If the integer is greater than the length of the original string, the new string will be left-justified with the optional string parameter taking the remaining space. If the string parameter is not given, the string will be padded with spaces.

```ruby
str ="abcd"
str.ljust(4)          => "abcd"
str.ljust(10)         => "abcd      "
```

To right-justify a string, use the `rjust` method. This takes in two parameters, an integer representing the number of characters of the new string and a string, representing the pattern to be filled.

If the integer is greater than the length of the original string, the new string will be right-justified with the optional string parameter taking the remaining space. If the string parameter is not given, the string will be padded with spaces.

```ruby
str = "abcd"
str.rjust(4)          => "abcd"
str.rjust(10)         => "      abcd"
```

To center a string, use the `center` method. This takes in two parameters, an integer representing the width of the new string and a string, which the original string will be padded with. The string will be aligned to the center.

```ruby
str = "abcd"
str.center(4)          => "abcd"
str.center(10)         => "   abcd   "
```

Section 11.6: Splitting a String

`String#split` splits a `String` into an `Array`, based on a delimiter.

```
"alpha,beta".split(",")
# => ["alpha", "beta"]
```

An empty `String` results into an empty `Array`:

```
"".split(",")
# => []
```

A non-matching delimiter results in an `Array` containing a single item:

```
"alpha,beta".split(".")
# => ["alpha,beta"]
```

You can also split a string using regular expressions:

```
"alpha, beta,gamma".split(/, ?/)
# => ["alpha", "beta", "gamma"]
```

The delimiter is optional, by default a string is split on whitespace:

```
"alpha beta".split
# => ["alpha", "beta"]
```

Section 11.7: String starts with

To find if a string starts with a pattern, the `start_with?` method comes in handy

```
str = "zebras are cool"
str.start_with?("zebras")      => true
```

You can also check the position of the pattern with `index`

```
str = "zebras are cool"
str.index("zebras").zero?      => true
```

Section 11.8: Joining Strings

`Array#join` joins an `Array` into a `String`, based on a delimiter:

```
["alpha", "beta"].join(",")
# => "alpha,beta"
```

The delimiter is optional, and defaults to an empty `String`.

```
["alpha", "beta"].join
# => "alphabeta"
```

An empty `Array` results in an empty `String`, no matter which delimiter is used.

```
[].join(",")
```

```
# => ""
```

Section 11.9: String interpolation

The double-quoted delimiter `"` and `%Q` sequence supports string interpolation using `#{ruby_expression}`:

```
puts "Now is #{Time.now}"
# Now is Now is 2016-07-21 12:47:45 +0200

puts %Q(Now is #{Time.now})
# Now is Now is 2016-07-21 12:47:45 +0200
```

Section 11.10: String ends with

To find if a string ends with a pattern, the `end_with?` method comes in handy

```
str = "I like pineapples"
str.end_with?("pineaaples")      => false
```

Section 11.11: Formatted strings

Ruby can inject an array of values into a string by replacing any placeholders with the values from the supplied array.

```
"Hello %s, my name is %s!" % ['World', 'br3nt']
# => Hello World, my name is br3nt!
```

The place holders are represented by two `%s` and the values are supplied by the array `['Hello', 'br3nt']`. The `%` operator instructs the string to inject the values of the array.

Section 11.12: String Substitution

```
p "This is %s" % "foo"
# => "This is foo"

p "%s %s %s" % ["foo", "bar", "baz"]
# => "foo bar baz"

p "%{foo} == %{foo}" % {:foo => "foo" }
# => "foo == foo"
```

See String `%` docs and Kernel::sprintf for more details.

Section 11.13: Multiline strings

The easiest way to create a multiline string is to just use multiple lines between quotation marks:

```
address = "Four score and seven years ago our fathers brought forth on this
continent, a new nation, conceived in Liberty, and dedicated to the
proposition that all men are created equal."
```

The main problem with that technique is that if the string includes a quotation, it'll break the string syntax. To work around the problem, you can use a heredoc instead:

```
puts <<-RAVEN
  Once upon a midnight dreary, while I pondered, weak and weary,
  Over many a quaint and curious volume of forgotten lore—
      While I nodded, nearly napping, suddenly there came a tapping,
  As of some one gently rapping, rapping at my chamber door.
  "'Tis some visitor," I muttered, "tapping at my chamber door—
              Only this and nothing more."
  RAVEN
```

Ruby supports shell-style here documents with `<<EOT`, but the terminating text must start the line. That screws up code indentation, so there's not a lot of reason to use that style. Unfortunately, the string will have indentations depending no how the code itself is indented.

Ruby 2.3 solves the problem by introducing `<<~` which strips out excess leading spaces:

Version ≥ 2.3

```
def build_email(address)
  return (<<~EMAIL)
    TO: #{address}

    To Whom It May Concern:

    Please stop playing the bagpipes at sunrise!

    Regards,
    Your neighbor
  EMAIL
end
```

Percent Strings also work to create multiline strings:

```
%q(
HAMLET        Do you see yonder cloud that's almost in shape of a camel?
POLONIUS        By the mass, and 'tis like a camel, indeed.
HAMLET        Methinks it is like a weasel.
POLONIUS        It is backed like a weasel.
HAMLET        Or like a whale?
POLONIUS        Very like a whale
)
```

There are a few ways to avoid interpolation and escape sequences:

- Single quote instead of double quote: `'\n is a carriage return.'`
- Lower case q in a percent string: `%q[#{not-a-variable}]`
- Single quote the terminal string in a heredoc:

```
<<-'CODE'
  puts 'Hello world!'
CODE
```

Section 11.14: String character replacements

The `tr` method returns a copy of a string where the characters of the first argument are replaced by the characters of the second argument.

```ruby
"string".tr('r', 'l') # => "stling"
```

To replace only the first occurrence of a pattern with with another expression use the `sub` method

```ruby
"string ring".sub('r', 'l') # => "stling ring"
```

If you would like to replace *all* occurrences of a pattern with that expression use `gsub`

```ruby
"string ring".gsub('r','l') # => "stling ling"
```

To delete characters, pass in an empty string for the second parameter

You can also use regular expressions in all these methods.

It's important to note that these methods will only return a new copy of a string and won't modify the string in place. To do that, you need to use the `tr!`, `sub!` and `gsub!` methods respectively.

Section 11.15: Understanding the data in a string

In Ruby, a string is just a sequence of bytes along with the name of an encoding (such as `UTF-8`, `US-ASCII`, `ASCII-8BIT`) that specifies how you might interpret those bytes as characters.

Ruby strings can be used to hold text (basically a sequence of characters), in which case the UTF-8 encoding is usually used.

```ruby
"abc".bytes  # => [97, 98, 99]
"abc".encoding.name  # => "UTF-8"
```

Ruby strings can also be used to hold binary data (a sequence of bytes), in which case the ASCII-8BIT encoding is usually used.

```ruby
[42].pack("i").encoding  # => "ASCII-8BIT"
```

It is possible for the sequence of bytes in a string to not match the encoding, resulting in errors if you try to use the string.

```ruby
"\xFF \xFF".valid_encoding? # => false
"\xFF \xFF".split(' ')      # ArgumentError: invalid byte sequence in UTF-8
```

Chapter 12: DateTime

Section 12.1: DateTime from string

`DateTime.parse` is a very useful method which construct a DateTime from a string, guessing its format.

```
DateTime.parse('Jun, 8 2016')
# => #<DateTime: 2016-06-08T00:00:00+00:00 ((2457548j,0s,0n),+0s,2299161j)>
DateTime.parse('201603082330')
# => #<DateTime: 2016-03-08T23:30:00+00:00 ((2457456j,84600s,0n),+0s,2299161j)>
DateTime.parse('04-11-2016 03:50')
# => #<DateTime: 2016-11-04T03:50:00+00:00 ((2457697j,13800s,0n),+0s,2299161j)>
DateTime.parse('04-11-2016 03:50 -0300')
# => #<DateTime: 2016-11-04T03:50:00-03:00 ((2457697j,24600s,0n),-10800s,2299161j)
```

Note: There are lots of other formats that `parse` recognizes.

Section 12.2: New

```
DateTime.new(2014,10,14)
# => #<DateTime: 2014-10-14T00:00:00+00:00 ((2456945j,0s,0n),+0s,2299161j)>
```

Current time:

```
DateTime.now
# => #<DateTime: 2016-08-04T00:43:58-03:00 ((2457605j,13438s,667386397n),-10800s,2299161j)>
```

Note that it gives the current time in your timezone

Section 12.3: Add/subtract days to DateTime

`DateTime` + `Fixnum` (days quantity)

```
DateTime.new(2015,12,30,23,0) + 1
# => #<DateTime: 2015-12-31T23:00:00+00:00 ((2457388j,82800s,0n),+0s,2299161j)>
```

`DateTime` + `Float` (days quantity)

```
DateTime.new(2015,12,30,23,0) + 2.5
# => #<DateTime: 2016-01-02T11:00:00+00:00 ((2457390j,39600s,0n),+0s,2299161j)>
```

`DateTime` + `Rational` (days quantity)

```
DateTime.new(2015,12,30,23,0) + Rational(1,2)
# => #<DateTime: 2015-12-31T11:00:00+00:00 ((2457388j,39600s,0n),+0s,2299161j)>
```

`DateTime` - `Fixnum` (days quantity)

```
DateTime.new(2015,12,30,23,0) - 1
# => #<DateTime: 2015-12-29T23:00:00+00:00 ((2457388j,82800s,0n),+0s,2299161j)>
```

`DateTime` - `Float` (days quantity)

```
DateTime.new(2015,12,30,23,0) - 2.5
```

```
# => #<DateTime: 2015-12-28T11:00:00+00:00 ((2457385j,39600s,0n),+0s,2299161j)>
```

DateTime - **Rational** (days quantity)

```
DateTime.new(2015,12,30,23,0) - Rational(1,2)
# => #<DateTime: 2015-12-30T11:00:00+00:00 ((2457387j,39600s,0n),+0s,2299161j)>
```

Chapter 13: Time

Section 13.1: How to use the strftime method

Converting a time to a string is a pretty common thing to do in Ruby. `strftime` is the method one would use to convert time to a string.

Here are some examples:

```
Time.now.strftime("%Y-%m-d %H:%M:S") #=> "2016-07-27 08:45:42"
```

This can be simplified even further

```
Time.now.strftime("%F %X")  #=> "2016-07-27 08:45:42"
```

Section 13.2: Creating time objects

Get current time:

```
Time.now
Time.new # is equivalent if used with no parameters
```

Get specific time:

```
Time.new(2010, 3, 10) #10 March 2010 (Midnight)
Time.new(2015, 5, 3, 10, 14) #10:14 AM on 3 May 2015
Time.new(2050, "May", 3, 21, 8, 16, "+10:00") #09:08:16 PM on 3 May 2050
```

To convert a time to epoch you can use the `to_i` method:

```
Time.now.to_i # => 1478633386
```

You can also convert back from epoch to Time using the `at` method:

```
Time.at(1478633386) # => 2016-11-08 17:29:46 -0200
```

Chapter 14: Numbers

Section 14.1: Converting a String to Integer

You can use the `Integer` method to convert a `String` to an `Integer`:

```
Integer("123")      # => 123
Integer("0xFF")     # => 255
Integer("0b100")    # => 4
Integer("0555")     # => 365
```

You can also pass a base parameter to the `Integer` method to convert numbers from a certain base

```
Integer('10', 5)    # => 5
Integer('74', 8)    # => 60
Integer('NUM', 36)  # => 30910
```

Note that the method raises an `ArgumentError` if the parameter cannot be converted:

```
Integer("hello")
# raises ArgumentError: invalid value for Integer(): "hello"
Integer("23-hello")
# raises ArgumentError: invalid value for Integer(): "23-hello"
```

You can also use the `String#to_i` method. However, this method is slightly more permissive and has a different behavior than `Integer`:

```
"23".to_i         # => 23
"23-hello".to_i   # => 23
"hello".to_i      # => 0
```

`String#to_i` accepts an argument, the base to interpret the number as:

```
"10".to_i(2) # => 2
"10".to_i(3) # => 3
"A".to_i(16) # => 10
```

Section 14.2: Creating an Integer

```
0       # creates the Fixnum 0
123     # creates the Fixnum 123
1_000   # creates the Fixnum 1000. You can use _ as separator for readability
```

By default the notation is base 10. However, there are some other built-in notations for different bases:

```
0xFF    # Hexadecimal representation of 255, starts with a 0x
0b100   # Binary representation of 4, starts with a 0b
0555    # Octal representation of 365, starts with a 0 and digits
```

Section 14.3: Rounding Numbers

The `round` method will round a number up if the first digit after its decimal place is 5 or higher and round down if that digit is 4 or lower. This takes in an optional argument for the precision you're looking for.

```ruby
4.89.round        # => 5
4.25.round        # => 4
3.141526.round(1) # => 3.1
3.141526.round(2) # => 3.14
3.141526.round(4) # => 3.1415
```

Floating point numbers can also be rounded down to the highest integer lower than the number with the `floor` method

```ruby
4.9999999999999.floor # => 4
```

They can also be rounded up to the lowest integer higher than the number using the `ceil` method

```ruby
4.0000000000001.ceil  # => 5
```

Section 14.4: Even and Odd Numbers

The `even?` method can be used to determine if a number is even

```ruby
4.even?      # => true
5.even?      # => false
```

The `odd?` method can be used to determine if a number is odd

```ruby
4.odd?       # => false
5.odd?       # => true
```

Section 14.5: Rational Numbers

`Rational` represents a rational number as numerator and denominator:

```ruby
r1 = Rational(2, 3)
r2 = 2.5.to_r
r3 = r1 + r2
r3.numerator   # => 19
r3.denominator # => 6
Rational(2, 4) # => (1/2)
```

Other ways of creating a Rational

```ruby
Rational('2/3')  # => (2/3)
Rational(3)      # => (3/1)
Rational(3, -5)  # => (-3/5)
Rational(0.2)    # => (3602879701896397/18014398509481984)
Rational('0.2')  # => (1/5)
0.2.to_r         # => (3602879701896397/18014398509481984)
0.2.rationalize  # => (1/5)
'1/4'.to_r       # => (1/4)
```

Section 14.6: Complex Numbers

```ruby
1i     # => (0+1i)
1.to_c # => (1+0i)
rectangular = Complex(2, 3)  # => (2+3i)
polar       = Complex('1@2') # => (-0.4161468365471424+0.9092974268256817i)
```

```
polar.rectangular # => [-0.4161468365471424, 0.9092974268256817]
rectangular.polar # => [3.605551275463989, 0.982793723247329]
rectangular + polar # => (1.5838531634528576+3.909297426825682i)
```

Section 14.7: Converting a number to a string

Fixnum#to_s takes an optional base argument and represents the given number in that base:

```
2.to_s(2)   # => "10"
3.to_s(2)   # => "11"
3.to_s(3)   # => "10"
10.to_s(16) # => "a"
```

If no argument is provided, then it represents the number in base 10

```
2.to_s # => "2"
10423.to_s # => "10423"
```

Section 14.8: Dividing two numbers

When dividing two numbers pay attention to the type you want in return. Note that dividing **two integers will invoke the integer division**. If your goal is to run the float division, at least one of the parameters should be of `float` type.

Integer division:

```
3 / 2 # => 1
```

Float division

```
3 / 3.0 # => 1.0

16 / 2 / 2    # => 4
16 / 2 / 2.0  # => 4.0
16 / 2.0 / 2  # => 4.0
16.0 / 2 / 2  # => 4.0
```

Chapter 15: Symbols

Section 15.1: Creating a Symbol

The most common way to create a `Symbol` object is by prefixing the string identifier with a colon:

```
:a_symbol       # => :a_symbol
:a_symbol.class # => Symbol
```

Here are some alternative ways to define a `Symbol`, in combination with a `String` literal:

```
:"a_symbol"
"a_symbol".to_sym
```

Symbols also have a `%s` sequence that supports arbitrary delimiters similar to how `%q` and `%Q` work for strings:

```
%s(a_symbol)
%s{a_symbol}
```

The `%s` is particularly useful to create a symbol from an input that contains white space:

```
%s{a symbol} # => :"a symbol"
```

While some interesting symbols (`:/`, `:[]`, `:^`, etc.) can be created with certain string identifiers, note that symbols cannot be created using a numeric identifier:

```
:1 # => syntax error, unexpected tINTEGER, ...
:0.3 # => syntax error, unexpected tFLOAT, ...
```

Symbols may end with a single `?` or `!` without needing to use a string literal as the symbol's identifier:

```
:hello?  # :"hello?" is not necessary.
:world!  # :"world!" is not necessary.
```

Note that all of these different methods of creating symbols will return the same object:

```
:symbol.object_id == "symbol".to_sym.object_id
:symbol.object_id == %s{symbol}.object_id
```

Since Ruby 2.0 there is a shortcut for creating an array of symbols from words:

```
%i(numerator denominator) == [:numerator, :denominator]
```

Section 15.2: Converting a String to Symbol

Given a `String`:

```
s = "something"
```

there are several ways to convert it to a `Symbol`:

```
s.to_sym
# => :something
```

```
:"#{s}"
# => :something
```

Section 15.3: Converting a Symbol to String

Given a `Symbol`:

```
s = :something
```

The simplest way to convert it to a `String` is by using the `Symbol#to_s` method:

```
s.to_s
# => "something"
```

Another way to do it is by using the `Symbol#id2name` method which is an alias for the `Symbol#to_s` method. But it's a method that is unique to the `Symbol` class:

```
s.id2name
# => "something"
```

Chapter 16: Comparable

Parameter	Details
other	The instance to be compared to `self`

Section 16.1: Rectangle comparable by area

`Comparable` is one of the most popular modules in Ruby. Its purpose is to provide with convenience comparison methods.

To use it, you have to `include Comparable` and define the space-ship operator (`<=>`):

```
class Rectangle
  include Comparable

  def initialize(a, b)
    @a = a
    @b = b
  end

  def area
    @a * @b
  end

  def <=>(other)
    area <=> other.area
  end
end

r1 = Rectangle.new(1, 1)
r2 = Rectangle.new(2, 2)
r3 = Rectangle.new(3, 3)

r2 >= r1 # => true
r2.between? r1, r3 # => true
r3.between? r1, r2 # => false
```

Chapter 17: Control Flow

Section 17.1: if, elsif, else and end

Ruby offers the expected `if` and `else` expressions for branching logic, terminated by the `end` keyword:

```
# Simulate flipping a coin
result = [:heads, :tails].sample

if result == :heads
  puts 'The coin-toss came up "heads"'
else
  puts 'The coin-toss came up "tails"'
end
```

In Ruby, `if` statements are expressions that evaluate to a value, and the result can be assigned to a variable:

```
status = if age < 18
           :minor
         else
           :adult
         end
```

Ruby also offers C-style ternary operators (see here for details) that can be expressed as:

```
some_statement ? if_true : if_false
```

This means the above example using if-else can also be written as

```
status = age < 18 ? :minor : :adult
```

Additionally, Ruby offers the `elsif` keyword which accepts an expression to enables additional branching logic:

```
label = if shirt_size == :s
          'small'
        elsif shirt_size == :m
          'medium'
        elsif shirt_size == :l
          'large'
        else
          'unknown size'
        end
```

If none of the conditions in an `if`/`elsif` chain are true, and there is no `else` clause, then the expression evaluates to nil. This can be useful inside string interpolation, since `nil.to_s` is the empty string:

```
"user#{'s' if @users.size != 1}"
```

Section 17.2: Case statement

Ruby uses the `case` keyword for switch statements.

As per the Ruby Docs:

> Case statements consist of an optional condition, which is in the position of an argument to `case`, and zero or more `when` clauses. The first `when` clause to match the condition (or to evaluate to Boolean truth, if the condition is null) "wins", and its code stanza is executed. The value of the case statement is the value of the successful `when` clause, or `nil` if there is no such clause.
>
> A case statement can end with an `else` clause. Each `when` a statement can have multiple candidate values, separated by commas.

Example:

```
case x
when 1,2,3
  puts "1, 2, or 3"
when 10
  puts "10"
else
  puts "Some other number"
end
```

Shorter version:

```
case x
when 1,2,3 then puts "1, 2, or 3"
when 10 then puts "10"
else puts "Some other number"
end
```

The value of the `case` clause is matched with each `when` clause using the `===` method (not `==`). Therefore it can be used with a variety of different types of objects.

A `case` statement can be used with Ranges:

```
case 17
when 13..19
  puts "teenager"
end
```

A `case` statement can be used with a Regexp:

```
case "google"
when /oo/
  puts "word contains oo"
end
```

A `case` statement can be used with a Proc or lambda:

```
case 44
when -> (n) { n.even? or n < 0 }
  puts "even or less than zero"
end
```

A `case` statement can be used with Classes:

```
case x
when Integer
  puts "It's an integer"
```

```ruby
when String
  puts "It's a string"
end
```

By implementing the `===` method you can create your own match classes:

```ruby
class Empty
  def self.===(object)
    !object or "" == object
  end
end

case ""
when Empty
  puts "name was empty"
else
  puts "name is not empty"
end
```

A `case` statement can be used without a value to match against:

```ruby
case
when ENV['A'] == 'Y'
  puts 'A'
when ENV['B'] == 'Y'
  puts 'B'
else
  puts 'Neither A nor B'
end
```

A `case` statement has a value, so you can use it as a method argument or in an assignment:

```ruby
description = case 16
              when 13..19 then "teenager"
              else ""
              end
```

Section 17.3: Truthy and Falsy values

In Ruby, there are exactly two values which are considered "falsy", and will return false when tested as a condition for an `if` expression. They are:

- `nil`
- boolean `false`

All other values are considered "truthy", including:

- `0` - numeric zero (Integer or otherwise)
- `""` - Empty strings
- `"\n"` - Strings containing only whitespace
- `[]` - Empty arrays
- `{}` - Empty hashes

Take, for example, the following code:

```ruby
def check_truthy(var_name, var)
  is_truthy = var ? "truthy" : "falsy"
```

```
  puts "#{var_name} is #{is_truthy}"
end

check_truthy("false", false)
check_truthy("nil", nil)
check_truthy("0", 0)
check_truthy("empty string", "")
check_truthy("\\n", "\n")
check_truthy("empty array", [])
check_truthy("empty hash", {})
```

Will output:

```
false is falsy
nil is falsy
0 is truthy
empty string is truthy
\n is truthy
empty array is truthy
empty hash is truthy
```

Section 17.4: Inline if/unless

A common pattern is to use an inline, or trailing, `if` or `unless`:

```
puts "x is less than 5" if x < 5
```

This is known as a conditional *modifier*, and is a handy way of adding simple guard code and early returns:

```
def save_to_file(data, filename)
  raise "no filename given" if filename.empty?
  return false unless data.valid?

  File.write(filename, data)
end
```

It is not possible to add an `else` clause to these modifiers. Also it is generally not recommended to use conditional modifiers inside the main logic -- For complex code one should use normal `if`, `elsif`, `else` instead.

Section 17.5: while, until

A `while` loop executes the block while the given condition is met:

```
i = 0
while i < 5
  puts "Iteration ##{i}"
  i +=1
end
```

An `until` loop executes the block while the conditional is false:

```
i = 0
until i == 5
  puts "Iteration ##{i}"
  i +=1
end
```

Section 17.6: Flip-Flop operator

The flip flop operator `..` is used between two conditions in a conditional statement:

```
(1..5).select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 3, 4]
```

The condition evaluates to `false` *until* the first part becomes `true`. Then it evaluates to `true` *until* the second part becomes `true`. After that it switches to `false` again.

This example illustrates what is being selected:

```
[1, 2, 2, 3, 4, 4, 5].select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 2, 3, 4]
```

The flip-flop operator only works inside ifs (including `unless`) and ternary operator. Otherwise it is being considered as the range operator.

```
(1..5).select do |e|
  (e == 2) .. (e == 4)
end
# => ArgumentError: bad value for range
```

It can switch from `false` to `true` and backwards multiple times:

```
((1..5).to_a * 2).select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 3, 4, 2, 3, 4]
```

Section 17.7: Or-Equals/Conditional assignment operator (||=)

Ruby has an or-equals operator that allows a value to be assigned to a variable if and only if that variable evaluates to either `nil` or `false`.

```
||= # this is the operator that achieves this.
```

this operator with the double pipes representing or and the equals sign representing assigning of a value. You may think it represents something like this:

```
x = x || y
```

this above example is not correct. The or-equals operator actually represents this:

```
x || x = y
```

If `x` evaluates to `nil` or `false` then `x` is assigned the value of `y`, and left unchanged otherwise.

Here is a practical use-case of the or-equals operator. Imagine you have a portion of your code that is expected to send an email to a user. What do you do if for what ever reason there is no email for this user. You might write something like this:

```ruby
if user_email.nil?
   user_email = "error@yourapp.com"
end
```

Using the or-equals operator we can cut this entire chunk of code, providing clean, clear control and functionality.

```ruby
user_email ||= "error@yourapp.com"
```

In cases where `false` is a valid value, care must be taken to not override it accidentally:

```ruby
has_been_run = false
has_been_run ||= true
#=> true

has_been_run = false
has_been_run = true if has_been_run.nil?
#=> false
```

Section 17.8: unless

A common statement is `if !(some condition)`. Ruby offers the alternative of the `unless` statement.

The structure is exactly the same as an `if` statement, except the condition is negative. Also, the `unless` statement does not support `elsif`, but it does support `else`:

```ruby
# Prints not inclusive
unless 'hellow'.include?('all')
  puts 'not inclusive'
end
```

Section 17.9: throw, catch

Unlike many other programming languages, the `throw` and `catch` keywords are not related to exception handling in Ruby.

In Ruby, `throw` and `catch` act a bit like labels in other languages. They are used to change the control flow, but are not related to a concept of "error" like Exceptions are.

```ruby
catch(:out) do
  catch(:nested) do
    puts "nested"
  end

  puts "before"
  throw :out
  puts "will not be executed"
end
puts "after"
# prints "nested", "before", "after"
```

Section 17.10: Ternary operator

Ruby has a ternary operator (`?:`), which returns one of two value based on if a condition evaluates as truthy:

```ruby
conditional ? value_if_truthy : value_if_falsy
```

```ruby
value = true
value ? "true" : "false"
#=> "true"

value = false
value ? "true" : "false"
#=> "false"
```

it is the same as writing `if a then b else c end`, though the ternary is preferred

Examples:

```ruby
puts (if 1 then 2 else 3 end) # => 2

puts 1 ? 2 : 3                # => 2

x = if 1 then 2 else 3 end
puts x                        # => 2
```

Section 17.11: Loop control with break, next, and redo

The flow of execution of a Ruby block may be controlled with the `break`, `next`, and `redo` statements.

`break`

The `break` statement will exit the block immediately. Any remaining instructions in the block will be skipped, and the iteration will end:

```ruby
actions = %w(run jump swim exit macarena)
index = 0

while index < actions.length
  action = actions[index]

  break if action == "exit"

  index += 1
  puts "Currently doing this action: #{action}"
end

# Currently doing this action: run
# Currently doing this action: jump
# Currently doing this action: swim
```

`next`

The `next` statement will return to the top of the block immediately, and proceed with the next iteration. Any remaining instructions in the block will be skipped:

```ruby
actions = %w(run jump swim rest macarena)
index = 0

while index < actions.length
  action = actions[index]
  index += 1

  next if action == "rest"

  puts "Currently doing this action: #{action}"
```

```
end

# Currently doing this action: run
# Currently doing this action: jump
# Currently doing this action: swim
# Currently doing this action: macarena
```

`redo`

The `redo` statement will return to the top of the block immediately, and retry the same iteration. Any remaining instructions in the block will be skipped:

```
actions = %w(run jump swim sleep macarena)
index = 0
repeat_count = 0

while index < actions.length
  action = actions[index]
  puts "Currently doing this action: #{action}"

  if action == "sleep"
    repeat_count += 1
    redo if repeat_count < 3
  end

  index += 1
end

# Currently doing this action: run
# Currently doing this action: jump
# Currently doing this action: swim
# Currently doing this action: sleep
# Currently doing this action: sleep
# Currently doing this action: sleep
# Currently doing this action: macarena
```

`Enumerable` iteration

In addition to loops, these statements work with Enumerable iteration methods, such as `each` and `map`:

```
[1, 2, 3].each do |item|
  next if item.even?
  puts "Item: #{item}"
end

# Item: 1
# Item: 3
```

Block result values

In both the `break` and `next` statements, a value may be provided, and will be used as a block result value:

```
even_value = for value in [1, 2, 3]
  break value if value.even?
end

puts "The first even value is: #{even_value}"
```

```
# The first even value is: 2
```

Section 17.12: return vs. next: non-local return in a block

Consider this *broken* snippet:

```
def foo
  bar = [1, 2, 3, 4].map do |x|
    return 0 if x.even?
    x
  end
  puts 'baz'
  bar
end
foo # => 0
```

One might expect `return` to yield a value for `map`'s array of block results. So the return value of `foo` would be `[1, 0, 3, 0]`. Instead, `return` **returns a value from the method** `foo`. Notice that `baz` isn't printed, which means execution never reached that line.

`next` with a value does the trick. It acts as a block-level `return`.

```
def foo
  bar = [1, 2, 3, 4].map do |x|
    next 0 if x.even?
    x
  end
  puts 'baz'
  bar
end
foo # baz
    # => [1, 0, 3, 0]
```

In the absence of a `return`, the value returned by the block is the value of its last expression.

Section 17.13: begin, end

The `begin` block is a control structure that groups together multiple statements.

```
begin
  a = 7
  b = 6
  a * b
end
```

A `begin` block will return the value of the last statement in the block. The following example will return `3`.

```
begin
  1
  2
  3
end
```

The `begin` block is useful for conditional assignment using the `||=` operator where multiple statements may be required to return a result.

```
circumference ||=
```

```ruby
begin
  radius = 7
  tau = Math::PI * 2
  tau * radius
end
```

It can also be combined with other block structures such as `rescue`, `ensure`, `while`, `if`, `unless`, etc to provide greater control of program flow.

`Begin` blocks are not code blocks, like `{ ... }` or `do ... end`; they cannot be passed to functions.

Section 17.14: Control flow with logic statements

While it might seem counterintuitive, you can use logical operators to determine whether or not a statement is run. For instance:

```ruby
File.exist?(filename) or STDERR.puts "#{filename} does not exist!"
```

This will check to see if the file exists and only print the error message if it doesn't. The `or` statement is lazy, which means it'll stop executing once it's sure which whether it's value is true or false. As soon as the first term is found to be true, there's no need to check the value of the other term. But if the first term is false, it must check the second term.

A common use is to set a default value:

```ruby
glass = glass or 'full' # Optimist!
```

That sets the value of `glass` to 'full' if it's not already set. More concisely, you can use the symbolic version of `or`:

```ruby
glass ||= 'empty' # Pessimist.
```

It's also possible to run the second statement only if the first one is false:

```ruby
File.exist?(filename) and puts "#{filename} found!"
```

Again, `and` is lazy so it will only execute the second statement if necessary to arrive at a value.

The `or` operator has lower precedence than `and`. Similarly, `||` has lower precedence than `&&`. The symbol forms have higher precedence than the word forms. This is handy to know when you want to mix this technique with assignment:

```ruby
a = 1 and b = 2
#=> a==1
#=> b==2
```

```ruby
a = 1 && b = 2; puts a, b
#=> a==2
#=> b==2
```

Note that the Ruby Style Guide recommends:

> The `and` and `or` keywords are banned. The minimal added readability is just not worth the high probability of introducing subtle bugs. For boolean expressions, always use `&&` and `||` instead. For flow control, use `if` and `unless`; `&&` and `||` are also acceptable but less clear.

Chapter 18: Methods

Functions in Ruby provide organized, reusable code to preform a set of actions. Functions simplify the coding process, prevent redundant logic, and make code easier to follow. This topic describes the declaration and utilization of functions, arguments, parameters, yield statements and scope in Ruby.

Section 18.1: Defining a method

Methods are defined with the `def` keyword, followed by the *method name* and an optional list of *parameter names* in parentheses. The Ruby code between `def` and `end` represents the *body* of the method.

```
def hello(name)
  "Hello, #{name}"
end
```

A method invocation specifies the method name, the object on which it is to be invoked (sometimes called the receiver), and zero or more argument values that are assigned to the named method parameters.

```
hello("World")
# => "Hello, World"
```

When the receiver is not explicit, it is `self`.

Parameter names can be used as variables within the method body, and the values of these named parameters come from the arguments to a method invocation.

```
hello("World")
# => "Hello, World"
hello("All")
# => "Hello, All"
```

Section 18.2: Yielding to blocks

You can send a block to your method and it can call that block multiple times. This can be done by sending a proc/lambda or such, but is easier and faster with `yield`:

```
def simple(arg1,arg2)
  puts "First we are here:  #{arg1}"
  yield
  puts "Finally we are here:  #{arg2}"
  yield
end
simple('start','end') { puts "Now we are inside the yield" }

#> First we are here:  start
#> Now we are inside the yield
#> Finally we are here:  end
#> Now we are inside the yield
```

Note that the `{ puts ... }` is not inside the parentheses, it implicitly comes after. This also means we can only have one `yield` block. We can pass arguments to the `yield`:

```
def simple(arg)
  puts "Before yield"
  yield(arg)
```

```
  puts "After yield"
end
simple('Dave') { |name| puts "My name is #{name}" }

#> Before yield
#> My name is Dave
#> After yield
```

With yield we can easily make iterators or any functions that work on other code:

```
def countdown(num)
  num.times do |i|
    yield(num-i)
  end
end
```

```
countdown(5) { |i| puts "Call number #{i}" }
```

```
#> Call number 5
#> Call number 4
#> Call number 3
#> Call number 2
#> Call number 1
```

In fact, it is with `yield` that things like `foreach`, `each` and `times` are generally implemented in classes.

If you want to find out if you have been given a block or not, use `block_given?`:

```
class Employees
  def names
    ret = []
    @employees.each do |emp|
      if block_given?
        yield(emp.name)
      else
        ret.push(emp.name)
      end
    end

    ret
  end
end
```

This example assumes that the `Employees` class has an `@employees` list that can be iterated with `each` to get objects that have employee names using the `name` method. If we are given a block, then we'll `yield` the name to the block, otherwise we just push it to an array that we return.

Section 18.3: Default parameters

```
def make_animal_sound(sound = 'Cuack')
    puts sound
end
```

```
make_animal_sound('Mooo') # Mooo
make_animal_sound         # Cuack
```

It's possible to include defaults for multiple arguments:

```ruby
def make_animal_sound(sound = 'Cuack', volume = 11)
    play_sound(sound, volume)
end

make_animal_sound('Mooo') # Spinal Tap cow
```

However, it's not possible to supply the second without also supplying the first. Instead of using positional parameters, try keyword parameters:

```ruby
def make_animal_sound(sound: 'Cuack', volume: 11)
    play_sound(sound, volume)
end

make_animal_sound(volume: 1) # Duck whisper
```

Or a hash parameter that stores options:

```ruby
def make_animal_sound(options = {})
    options[:sound]  ||= 'Cuak'
    options[:volume] ||= 11
    play_sound(sound, volume)
end

make_animal_sound(:sound => 'Mooo')
```

Default parameter values can be set by any ruby expression. The expression will run in the context of the method, so you can even declare local variables here. Note, won't get through code review. Courtesy of caius for pointing this out.

```ruby
def make_animal_sound( sound = ( raise 'TUU-too-TUU-too...' ) ); p sound; end

make_animal_sound 'blaaaa' # => 'blaaaa'
make_animal_sound          # => TUU-too-TUU-too... (RuntimeError)
```

Section 18.4: Optional parameter(s) (splat operator)

```ruby
def welcome_guests(*guests)
    guests.each { |guest| puts "Welcome #{guest}!" }
end
```

```ruby
welcome_guests('Tom')    # Welcome Tom!
welcome_guests('Rob', 'Sally', 'Lucas') # Welcome Rob!
                                        # Welcome Sally!
                                        # Welcome Lucas!
```

Note that `welcome_guests(['Rob', 'Sally', 'Lucas'])` will output `Welcome ["Rob", "Sally", "Lucas"]`! Instead, if you have a list, you can do `welcome_guests(*['Rob', 'Sally', 'Lucas'])` and that will work as `welcome_guests('Rob', 'Sally', 'Lucas')`.

Section 18.5: Required default optional parameter mix

```ruby
def my_mix(name,valid=true, *opt)
    puts name
    puts valid
    puts opt
```

```
end
```

Call as follows:

```
my_mix('me')
# 'me'
# true
# []

my_mix('me', false)
# 'me'
# false
# []

my_mix('me', true, 5, 7)
# 'me'
# true
# [5,7]
```

Section 18.6: Use a function as a block

Many functions in Ruby accept a block as an argument. E.g.:

```
[0, 1, 2].map {|i| i + 1}
 => [1, 2, 3]
```

If you already have a function that does what you want, you can turn it into a block using `&method(:fn)`:

```
def inc(num)
   num + 1
end

[0, 1, 2].map &method(:inc)
 => [1, 2, 3]
```

Section 18.7: Single required parameter

```
def say_hello_to(name)
    puts "Hello #{name}"
end
```

```
say_hello_to('Charles')    # Hello Charles
```

Section 18.8: Tuple Arguments

A method can take an array parameter and destructure it immediately into named local variables. Found on Mathias Meyer's blog.

```
def feed( amount, (animal, food) )

    p "#{amount} #{animal}s chew some #{food}"

end

feed 3, [ 'rabbit', 'grass' ] # => "3 rabbits chew some grass"
```

Section 18.9: Capturing undeclared keyword arguments (double splat)

The `**` operator works similarly to the `*` operator but it applies to keyword parameters.

```
def options(required_key:, optional_key: nil, **other_options)
  other_options
end

options(required_key: 'Done!', foo: 'Foo!', bar: 'Bar!')
#> { :foo => "Foo!", :bar => "Bar!" }
```

In the above example, if the `**other_options` is not used, an `ArgumentError: unknown keyword: foo, bar` error would be raised.

```
def without_double_splat(required_key:, optional_key: nil)
  # do nothing
end

without_double_splat(required_key: 'Done!', foo: 'Foo!', bar: 'Bar!')
#> ArgumentError: unknown keywords: foo, bar
```

This is handy when you have a hash of options that you want to pass to a method and you do not want to filter the keys.

```
def options(required_key:, optional_key: nil, **other_options)
  other_options
end

my_hash = { required_key: true, foo: 'Foo!', bar: 'Bar!' }

options(my_hash)
#> { :foo => "Foo!", :bar => "Bar!" }
```

It is also possible to *unpack* a hash using the `**` operator. This allows you to supply keyword directly to a method in addition to values from other hashes:

```
my_hash = { foo: 'Foo!', bar: 'Bar!' }

options(required_key: true, **my_hash)
#> { :foo => "Foo!", :bar => "Bar!" }
```

Section 18.10: Multiple required parameters

```
def greet(greeting, name)
    puts "#{greeting} #{name}"
end
```

```
greet('Hi', 'Sophie')    # Hi Sophie
```

Section 18.11: Method Definitions are Expressions

Defining a method in Ruby 2.x returns a symbol representing the name:

```
class Example
```

```ruby
  puts def hello
  end
end

#=> :hello
```

This allows for interesting metaprogramming techniques. For instance, methods can be wrapped by other methods:

```ruby
class Class
  def logged(name)
    original_method = instance_method(name)
    define_method(name) do |*args|
      puts "Calling #{name} with #{args.inspect}."
      original_method.bind(self).call(*args)
      puts "Completed #{name}."
    end
  end
end

class Meal
  def initialize
    @food = []
  end

  logged def add(item)
    @food << item
  end
end

meal = Meal.new
meal.add "Coffee"
# Calling add with ["Coffee"].
# Completed add.
```

Chapter 19: Hashes

A Hash is a dictionary-like collection of unique keys and their values. Also called associative arrays, they are similar to Arrays, but where an Array uses integers as its index, a Hash allows you to use any object type. You retrieve or create a new entry in a Hash by referring to its key.

Section 19.1: Creating a hash

A hash in Ruby is an object that implements a hash table, mapping keys to values. Ruby supports a specific literal syntax for defining hashes using `{}`:

```
my_hash = {}  # an empty hash
grades = { 'Mark' => 15, 'Jimmy' => 10, 'Jack' => 10 }
```

A hash can also be created using the standard `new` method:

```
my_hash = Hash.new  # any empty hash
my_hash = {}        # any empty hash
```

Hashes can have values of any type, including complex types like arrays, objects and other hashes:

```
mapping = { 'Mark' => 15, 'Jimmy' => [3,4], 'Nika' => {'a' => 3, 'b' => 5} }
mapping['Mark']   # => 15
mapping['Jimmy']  # => [3, 4]
mapping['Nika']   # => {"a"=>3, "b"=>5}
```

Also keys can be of any type, including complex ones:

```
mapping = { 'Mark' => 15, 5 => 10, [1, 2] => 9 }
mapping['Mark']  # => 15
mapping[[1, 2]]  # => 9
```

Symbols are commonly used as hash keys, and Ruby 1.9 introduced a new syntax specifically to shorten this process. The following hashes are equivalent:

```
# Valid on all Ruby versions
grades = { :Mark => 15, :Jimmy => 10, :Jack => 10 }
# Valid in Ruby version 1.9+
grades = { Mark: 15, Jimmy: 10, Jack: 10 }
```

The following hash (valid in all Ruby versions) is *different*, because all keys are strings:

```
grades = { "Mark" => 15, "Jimmy" => 10, "Jack" => 10 }
```

While both syntax versions can be mixed, the following is discouraged.

```
mapping = { :length => 45, width: 10 }
```

With Ruby 2.2+, there is an alternative syntax for creating a hash with symbol keys (most useful if the symbol contains spaces):

```
grades = { "Jimmy Choo": 10, :"Jack Sparrow": 10 }
# => { :"Jimmy Choo" => 10, :"Jack Sparrow" => 10}
```

Section 19.2: Setting Default Values

By default, attempting to lookup the value for a key which does not exist will return `nil`. You can optionally specify some other value to return (or an action to take) when the hash is accessed with a non-existent key. Although this is referred to as "the default value", it need not be a single value; it could, for example, be a computed value such as the length of the key.

The default value of a hash can be passed to its constructor:

```
h = Hash.new(0)

h[:hi] = 1
puts h[:hi]  # => 1
puts h[:bye] # => 0 returns default value instead of nil
```

A default can also be specified on an already constructed Hash:

```
my_hash = { human: 2, animal: 1 }
my_hash.default = 0
my_hash[:plant] # => 0
```

It is important to note that the **default value is not copied** each time a new key is accessed, which can lead to surprising results when the default value is a reference type:

```
# Use an empty array as the default value
authors = Hash.new([])

# Append a book title
authors[:homer] << 'The Odyssey'

# All new keys map to a reference to the same array:
authors[:plato] # => ['The Odyssey']
```

To circumvent this problem, the Hash constructor accepts a block which is executed each time a new key is accessed, and the returned value is used as the default:

```
authors = Hash.new { [] }

# Note that we're using += instead of <<, see below
authors[:homer] += ['The Odyssey']
authors[:plato] # => []

authors # => {:homer=>["The Odyssey"]}
```

Note that above we had to use += instead of << because the default value is not automatically assigned to the hash; using << would have added to the array, but authors[:homer] would have remained undefined:

```
authors[:homer] << 'The Odyssey' # ['The Odyssey']
authors[:homer] # => []
authors # => {}
```

In order to be able to assign default values on access, as well as to compute more sophisticated defaults, the default block is passed both the hash and the key:

```
authors = Hash.new { |hash, key| hash[key] = [] }
```

```ruby
authors[:homer] << 'The Odyssey'
authors[:plato] # => []

authors # => {:homer=>["The Odyssey"], :plato=>[]}
```

You can also use a default block to take an action and/or return a value dependent on the key (or some other data):

```ruby
chars = Hash.new { |hash,key| key.length }

chars[:test] # => 4
```

You can even create more complex hashes:

```ruby
page_views = Hash.new { |hash, key| hash[key] = { count: 0, url: key } }
page_views["http://example.com"][:count] += 1
page_views # => {"http://example.com"=>{:count=>1, :url=>"http://example.com"}}
```

In order to set the default to a Proc on an *already-existing* hash, use `default_proc=`:

```ruby
authors = {}
authors.default_proc = proc { [] }

authors[:homer] += ['The Odyssey']
authors[:plato] # => []

authors # {:homer=>["The Odyssey"]}
```

Section 19.3: Accessing Values

Individual values of a hash are read and written using the `[]` and `[]=` methods:

```ruby
my_hash = { length: 4, width: 5 }

my_hash[:length] #=> => 4

my_hash[:height] = 9

my_hash #=> {:length => 4, :width => 5, :height => 9 }
```

By default, accessing a key which has not been added to the hash returns **`nil`**, meaning it is always safe to attempt to look up a key's value:

```ruby
my_hash = {}

my_hash[:age] # => nil
```

Hashes can also contain keys in strings. If you try to access them normally it will just return a **`nil`**, instead you access them by their string keys:

```ruby
my_hash = { "name" => "user" }

my_hash[:name]    # => nil
my_hash["name"]   # => user
```

For situations where keys are expected or required to exist, hashes have a `fetch` method which will raise an exception when accessing a key that does not exist:

```ruby
my_hash = {}

my_hash.fetch(:age) #=> KeyError: key not found: :age
```

`fetch` accepts a default value as its second argument, which is returned if the key has not been previously set:

```ruby
my_hash =  {}
my_hash.fetch(:age, 45) #=> => 45
```

`fetch` can also accept a block which is returned if the key has not been previously set:

```ruby
my_hash = {}
my_hash.fetch(:age) { 21 } #=> 21

my_hash.fetch(:age) do |k|
  puts "Could not find #{k}"
end

#=> Could not find age
```

Hashes also support a `store` method as an alias for `[]=`:

```ruby
my_hash = {}

my_hash.store(:age, 45)

my_hash #=> { :age => 45 }
```

You can also get all values of a hash using the `values` method:

```ruby
my_hash = { length: 4, width: 5 }

my_hash.values #=> [4, 5]
```

Note: This is only for Ruby 2.3+ *#dig* is handy for nested **Hash**s. Extracts the nested value specified by the sequence of idx objects by calling dig at each step, returning nil if any intermediate step is nil.

```ruby
h = { foo: {bar: {baz: 1}}}

h.dig(:foo, :bar, :baz)   # => 1
h.dig(:foo, :zot, :xyz)   # => nil

g = { foo: [10, 11, 12] }
g.dig(:foo, 1)            # => 11
```

Section 19.4: Automatically creating a Deep Hash

Hash has a default value for keys that are requested but don't exist (nil):

```ruby
a = {}
p a[ :b ] # => nil
```

When creating a new Hash, one can specify the default:

```ruby
b = Hash.new 'puppy'
p b[ :b ]            # => 'puppy'
```

Hash.new also takes a block, which allows you to automatically create nested hashes, such as Perl's autovivification behavior or `mkdir -p`:

```
# h is the hash you're creating, and k the key.
#
hash = Hash.new { |h, k| h[k] = Hash.new &h.default_proc }
hash[ :a ][ :b ][ :c ] = 3

p hash # => { a: { b: { c: 3 } } }
```

Section 19.5: Iterating Over a Hash

A `Hash` includes the `Enumerable` module, which provides several iteration methods, such as: `Enumerable#each`, `Enumerable#each_pair`, `Enumerable#each_key`, and `Enumerable#each_value`.

`.each` and `.each_pair` iterate over each key-value pair:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each do |key, value|
    puts "#{key} = #{value}"
end

# => first_name = John
#    last_name = Doe
```

`.each_key` iterates over the keys only:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_key do |key|
  puts key
end

# => first_name
#    last_name
```

`.each_value` iterates over the values only:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_value do |value|
    puts value
end

# => John
#    Doe
```

`.each_with_index` iterates over the elements and provides the index of the iteration:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_with_index do |(key, value), index|
    puts "index: #{index} | key: #{key} | value: #{value}"
end

# => index: 0 | key: first_name | value: John
#    index: 1 | key: last_name | value: Doe
```

Section 19.6: Filtering hashes

`SELECT` returns a new `Hash` with key-value pairs for which the block evaluates to `true`.

```
{ :a => 1, :b => 2, :c => 3 }.select { |k, v| k != :a && v.even? } # => { :b => 2 }
```

When you will not need the *key* or *value* in a filter block, the convention is to use an `_` in that place:

```
{ :a => 1, :b => 2, :c => 3 }.select { |_, v| v.even? } # => { :b => 2 }
{ :a => 1, :b => 2, :c => 3 }.select { |k, _| k == :c } # => { :c => 3 }
```

`reject` returns a new `Hash` with key-value pairs for which the block evaluates to `false`:

```
{ :a => 1, :b => 2, :c => 3 }.reject { |_, v| v.even? } # => { :a => 1, :c => 3 }
{ :a => 1, :b => 2, :c => 3 }.reject { |k, _| k == :b } # => { :a => 1, :c => 3 }
```

Section 19.7: Conversion to and from Arrays

Hashes can be freely converted to and from arrays. Converting a hash of key/value pairs into an array will produce an array containing nested arrays for pair:

```
{ :a => 1, :b => 2 }.to_a # => [[:a, 1], [:b, 2]]
```

In the opposite direction a Hash can be created from an array of the same format:

```
[[:x, 3], [:y, 4]].to_h # => { :x => 3, :y => 4 }
```

Similarly, Hashes can be initialized using `Hash[]` and a list of alternating keys and values:

```
Hash[:a, 1, :b, 2] # => { :a => 1, :b => 2 }
```

Or from an array of arrays with two values each:

```
Hash[ [[:x, 3], [:y, 4]] ] # => { :x => 3, :y => 4 }
```

Hashes can be converted back to an Array of alternating keys and values using `flatten()`:

```
{ :a => 1, :b => 2 }.flatten # => [:a, 1, :b, 2]
```

The easy conversion to and from an array allows `Hash` to work well with many `Enumerable` methods such as `collect` and `zip`:

```
Hash[('a'..'z').collect{ |c| [c, c.upcase] }] # => { 'a' => 'A', 'b' => 'B', ... }

people = ['Alice', 'Bob', 'Eve']
height = [5.7, 6.0, 4.9]
Hash[people.zip(height)] # => { 'Alice' => 5.7, 'Bob' => '6.0', 'Eve' => 4.9 }
```

Section 19.8: Overriding hash function

Ruby hashes use the methods `hash` and `eql?` to perform the hash operation and assign objects stored in the hash to internal hash bins. The default implementation of `hash` in Ruby is the murmur hash function over all member fields of the hashed object. To override this behavior it is possible to override `hash` and `eql?` methods.

As with other hash implementations, two objects a and b, will be hashed to the same bucket if `a.hash == b.hash` and will be deemed identical if `a.eql?(b)`. Thus, when reimplementing `hash` and `eql?` one should take care to ensure that if `a` and `b` are equal under `eql?` they must return the same `hash` value. Otherwise this might result in duplicate entries in a hash. Conversely, a poor choice in `hash` implementation might lead many objects to share the same hash bucket, effectively destroying the O(1) look-up time and causing O(n) for calling `eql?` on all objects.

In the example below only the instance of class `A` is stored as a key, as it was added first:

```
class A
  def initialize(hash_value)
    @hash_value = hash_value
  end
  def hash
    @hash_value # Return the value given externally
  end
  def eql?(b)
    self.hash == b.hash
  end
end

class B < A
end

a = A.new(1)
b = B.new(1)

h = {}
h[a] = 1
h[b] = 2

raise "error" unless h.size == 1
raise "error" unless h.include? b
raise "error" unless h.include? a
```

Section 19.9: Getting all keys or values of hash

```
{foo: 'bar', biz: 'baz'}.keys   # => [:foo, :biz]
{foo: 'bar', biz: 'baz'}.values # => ["bar", "baz"]
{foo: 'bar', biz: 'baz'}.to_a   # => [[:foo, "bar"], [:biz, "baz"]]
{foo: 'bar', biz: 'baz'}.each   #<Enumerator: {:foo=>"bar", :biz=>"baz"}:each>
```

Section 19.10: Modifying keys and values

You can create a new hash with the keys or values modified, indeed you can also add or delete keys, using inject (AKA, reduce). For example to produce a hash with stringified keys and upper case values:

```
fruit = { name: 'apple', color: 'green', shape: 'round' }
# => {:name=>"apple", :color=>"green", :shape=>"round"}

new_fruit = fruit.inject({}) { |memo, (k,v)| memo[k.to_s] = v.upcase; memo }

# => new_fruit is {"name"=>"APPLE", "color"=>"GREEN", "shape"=>"ROUND"}
```

Hash is an enumerable, in essence a collection of key/value pairs. Therefore is has methods such as `each`, `map` and `inject`.

For every key/value pair in the hash the given block is evaluated, the value of memo on the first run is the seed

value passed to `inject`, in our case an empty hash, `{}`. The value of `memo` for subsequent evaluations is the returned value of the previous blocks evaluation, this is why we modify `memo` by setting a key with a value and then return `memo` at the end. The return value of the final blocks evaluation is the return value of `inject`, in our case `memo`.

To avoid the having to provide the final value, you could use each_with_object instead:

```
new_fruit = fruit.each_with_object({}) { |(k,v), memo| memo[k.to_s] = v.upcase }
```

Or even map:

Version ≥ 1.8

```
new_fruit = Hash[fruit.map{ |k,v| [k.to_s, v.upcase] }]
```

(See this answer for more details, including how to manipulate hashes in place.)

Section 19.11: Set Operations on Hashes

- **Intersection of Hashes**

  To get the intersection of two hashes, return the shared keys the values of which are equal:

  ```
  hash1 = { :a => 1, :b => 2 }
  hash2 = { :b => 2, :c => 3 }
  hash1.select { |k, v| (hash2.include?(k) && hash2[k] == v) } # => { :b => 2 }
  ```

- **Union (merge) of hashes:**

  keys in a hash are unique, if a key occurs in both hashes which are to be merged, the one from the hash that `merge` is called on is overwritten:

  ```
  hash1 = { :a => 1, :b => 2 }
  hash2 = { :b => 4, :c => 3 }

  hash1.merge(hash2) # => { :a => 1, :b => 4, :c => 3 }
  hash2.merge(hash1) # => { :b => 2, :c => 3, :a => 1 }
  ```

Chapter 20: Blocks and Procs and Lambdas

Section 20.1: Lambdas

```
# lambda using the arrow syntax
hello_world = -> { 'Hello World!' }
hello_world[]
# 'Hello World!'

# lambda using the arrow syntax accepting 1 argument
hello_world = ->(name) { "Hello #{name}!" }
hello_world['Sven']
# "Hello Sven!"

the_thing = lambda do |magic, ohai, dere|
  puts "magic! #{magic}"
  puts "ohai #{dere}"
  puts "#{ohai} means hello"
end

the_thing.call(1, 2, 3)
# magic! 1
# ohai 3
# 2 means hello

the_thing.call(1, 2)
# ArgumentError: wrong number of arguments (2 for 3)

the_thing[1, 2, 3, 4]
# ArgumentError: wrong number of arguments (4 for 3)
```

You can also use `->` to create and `.()` to call lambda

```
the_thing = ->(magic, ohai, dere) {
  puts "magic! #{magic}"
  puts "ohai #{dere}"
  puts "#{ohai} means hello"
}

the_thing.(1, 2, 3)
# => magic! 1
# => ohai 3
# => 2 means hello
```

Here you can see that a lambda is almost the same as a proc. However, there are several caveats:

- The arity of a lambda's arguments are enforced; passing the wrong number of arguments to a lambda, will raise an `ArgumentError`. They can still have default parameters, splat parameters, etc.
- `return`ing from within a lambda returns from the lambda, while `return`ing from a proc returns out of the enclosing scope:

```
def try_proc
  x = Proc.new {
    return # Return from try_proc
  }
```

```
  x.call
  puts "After x.call" # this line is never reached
end

def try_lambda
  y = -> {
    return # return from y
  }
  y.call
  puts "After y.call" # this line is not skipped
end

try_proc # No output
try_lambda # Outputs "After y.call"
```

Section 20.2: Partial Application and Currying

Technically, Ruby doesn't have functions, but methods. However, a Ruby method behaves almost identically to functions in other language:

```
def double(n)
  n * 2
end
```

This normal method/function takes a parameter `n`, doubles it and returns the value. Now let's define a higher order function (or method):

```
def triple(n)
  lambda {3 * n}
end
```

Instead of returning a number, `triple` returns a method. You can test it using the Interactive Ruby Shell:

```
$ irb --simple-prompt
>> def double(n)
>>   n * 2
>> end
=> :double
>> def triple(n)
>>   lambda {3 * n}
>> end
=> :triple
>> double(2)
=> 4
>> triple(2)
=> #<Proc:0x007fd07f07bdc0@(irb):7 (lambda)>
```

If you want to actually get the tripled number, you need to call (or "reduce") the lambda:

```
triple_two = triple(2)
triple_two.call # => 6
```

Or more concisely:

```
triple(2).call
```

Currying and Partial Applications

This is not useful in terms of defining very basic functionality, but it is useful if you want to have methods/functions that are not instantly called or reduced. For example, let's say you want to define methods that add a number by a specific number (for example `add_one(2) = 3`). If you had to define a ton of these you could do:

```
def add_one(n)
  n + 1
end

def add_two(n)
  n + 2
end
```

However, you could also do this:

```
add = -> (a, b) { a + b }
add_one = add.curry.(1)
add_two = add.curry.(2)
```

Using lambda calculus we can say that `add` is `(λa.(λb.(a+b)))`. Currying is a way of *partially applying* `add`. So `add.curry.(1)`, is `(λa.(λb.(a+b)))(1)` which can be reduced to `(λb.(1+b))`. Partial application means that we passed one argument to `add` but left the other argument to be supplied later. The output is a specialized method.

More useful examples of currying

Let's say we have really big general formula, that if we specify certain arguments to it, we can get specific formulae from it. Consider this formula:

```
f(x, y, z) = sin(x\\*y)*sin(y\\*z)*sin(z\\*x)
```

This formula is made for working in three dimensions, but let's say we only want this formula with regards to y and z. Let's also say that to ignore x, we want to set it's value to pi/2. Let's first make the general formula:

```
f = ->(x, y, z) {Math.sin(x*y) * Math.sin(y*z) * Math.sin(z*x)}
```

Now, let's use currying to get our `yz` formula:

```
f_yz = f.curry.(Math::PI/2)
```

Then to call the lambda stored in `f_yz`:

```
f_xy.call(some_value_x, some_value_y)
```

This is pretty simple, but let's say we want to get the formula for `xz`. How can we set `y` to `Math::PI/2` if it's not the last argument? Well, it's a bit more complicated:

```
f_xz = -> (x,z) {f.curry.(x, Math::PI/2, z)}
```

In this case, we need to provide placeholders for the parameter we aren't pre-filling. For consistency we could write `f_xy` like this:

```
f_xy = -> (x,y) {f.curry.(x, y, Math::PI/2)}
```

Here's how the lambda calculus works for `f_yz`:

```
f = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x))))
f_yz = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x)))) (π/2) # Reduce =>
f_yz = (λy.(λz.(sin((π/2)*y) * sin(y*z) * sin(z*(π/2)))))
```

Now let's look at `f_xz`

```
f = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x))))
f_xz = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x)))) (λt.t) (π/2)  # Reduce =>
f_xz = (λt.(λz.(sin(t*(π/2)) * sin((π/2)*z) * sin(z*t))))
```

For more reading about lambda calculus try this.

Section 20.3: Objects as block arguments to methods

Putting a `&` (ampersand) in front of an argument will pass it as the method's block. Objects will be converted to a `Proc` using the `to_proc` method.

```
class Greeter
  def to_proc
    Proc.new do |item|
      puts "Hello, #{item}"
    end
  end
end

greet = Greeter.new

%w(world life).each(&greet)
```

This is a common pattern in Ruby and many standard classes provide it.

For example, `Symbol`s implement `to_proc` by sending themselves to the argument:

```
# Example implementation
class Symbol
  def to_proc
    Proc.new do |receiver|
      receiver.send self
    end
  end
end
```

This enables the useful `&:symbol` idiom, commonly used with `Enumerable` objects:

```
letter_counts = %w(just some words).map(&:length)  # [4, 4, 5]
```

Section 20.4: Converting to Proc

Objects that respond to `to_proc` can be converted to procs with the `&` operator (which will also allow them to be passed as blocks).

The class Symbol defines `#to_proc` so it tries to call the corresponding method on the object it receives as parameter.

```ruby
p [ 'rabbit', 'grass' ].map( &:upcase ) # => ["RABBIT", "GRASS"]
```

Method objects also define `#to_proc`.

```ruby
output = method( :p )

[ 'rabbit', 'grass' ].map( &output ) # => "rabbit\ngrass"
```

Section 20.5: Blocks

Blocks are chunks of code enclosed between braces {} (usually for single-line blocks) or `do..end` (used for multi-line blocks).

```ruby
5.times { puts "Hello world" } # recommended style for single line blocks

5.times do
    print "Hello "
    puts "world"
end   # recommended style for multi-line blocks

5.times {
    print "hello "
    puts "world" } # does not throw an error but is not recommended
```

Note: braces have higher precedence than `do..end`

Yielding

Blocks can be used inside methods and functions using the word `yield`:

```ruby
def block_caller
    puts "some code"
    yield
    puts "other code"
end
block_caller { puts "My own block" } # the block is passed as an argument to the method.
#some code
#My own block
#other code
```

Be careful though if `yield` is called without a block it will raise a `LocalJumpError`. For this purpose ruby provides another method called `block_given?` this allows you to check if a block was passed before calling yield

```ruby
def block_caller
  puts "some code"
  if block_given?
    yield
  else
    puts "default"
  end
  puts "other code"
end
block_caller
# some code
# default
# other code
block_caller { puts "not defaulted"}
# some code
```

```
# not defaulted
# other code
```

`yield` can offer arguments to the block as well

```
def yield_n(n)
  p = yield n if block_given?
  p || n
end
yield_n(12) {|n| n + 7 }
#=> 19
yield_n(4)
#=> 4
```

While this is a simple example `yield`ing can be very useful for allowing direct access to instance variables or evaluations inside the context of another object. For Example:

```
class Application
  def configuration
    @configuration ||= Configuration.new
    block_given? ? yield(@configuration) : @configuration
  end
end
class Configuration; end

app = Application.new
app.configuration do |config|
  puts config.class.name
end
# Configuration
#=> nil
app.configuration
#=> #<Configuration:0x2bf1d30>
```

As you can see using `yield` in this manner makes the code more readable than continually calling `app.configuration.#method_name`. Instead you can perform all the configuration inside the block keeping the code contained.

Variables

Variables for blocks are local to the block (similar to the variables of functions), they die when the block is executed.

```
my_variable = 8
3.times do |x|
    my_variable = x
    puts my_variable
end
puts my_variable
#=> 0
# 1
# 2
# 8
```

Blocks can't be saved, they die once executed. In order to save blocks you need to use `procs` and `lambdas`.

Chapter 21: Iteration

Section 21.1: Each

Ruby has many types of enumerators but the first and most simple type of enumerator to start with is `each`. We will print out `even` or `odd` for each number between `1` and `10` to show how `each` works.

Basically there are two ways to pass so called `blocks`. A `block` is a piece of code being passed which will be executed by the method which is called. The `each` method takes a `block` which it calls for every element of the collection of objects it was called on.

There are two ways to pass a block to a method:

Method 1: Inline
(1..10).each { |i| puts i.even? ? 'even' : 'odd' }

This is a very compressed and *ruby* way to solve this. Let's break this down piece by piece.

1. `(1..10)` is a range from `1` to `10` inclusive. If we wanted it to be `1` to `10` exclusive, we would write `(1...10)`.
2. `.each` is an enumerator that enumerates over `each` element in the object it is acting on. In this case, it acts on `each` number in the range.
3. `{ |i| puts i.even? ? 'even' : 'odd' }` is the block for the `each` statement, which itself can be broken down further.
    1. `|i|` this means that each element in the range is represented within the block by the identifier `i`.
    2. `puts` is an output method in Ruby that has an automatic line break after each time it prints. (We can use `print` if we don't want the automatic line break)
    3. `i.even?` checks if `i` is even. We could have also used `i % 2 == 0`; however, it is preferable to use built in methods.
    4. `? "even" : "odd"` this is ruby's ternary operator. The way a ternary operator is constructed is `expression ? a : b`. This is short for

    if expression a else b end

For code longer than one line the `block` should be passed as a `multiline block`.

Method 2: Multiline
(1..10).each do |i| if i.even? puts 'even' else puts 'odd' end end

In a `multiline block` the `do` replaces the opening bracket and `end` replaces the closing bracket from the `inline` style.

Ruby supports reverse_each as well. It will iterate the array backwards.

```
@arr = [1,2,3,4]
puts @arr.inspect # output is [1,2,3,4]

print "Reversed array elements["
@arr.reverse_each do |val|
        print " #{val} " # output is 4 3 2 1
end
print "]\n"
```

Section 21.2: Implementation in a class

`Enumerable` is the most popular module in Ruby. Its purpose is to provide you with iterable methods like `map`,

`SELECT`, `reduce`, etc. Classes that use `Enumerable` include `Array`, `Hash`, `Range`. To use it, you have to `include Enumerable` and implement `each`.

```
class NaturalNumbers
  include Enumerable

  def initialize(upper_limit)
    @upper_limit = upper_limit
  end

  def each(&block)
    0.upto(@upper_limit).each(&block)
  end
end

n = NaturalNumbers.new(6)

n.reduce(:+)                  # => 21
n.select(&:even?)             # => [0, 2, 4, 6]
n.map { |number| number ** 2 } # => [0, 1, 4, 9, 16, 25, 36]
```

Section 21.3: Iterating over complex objects

Arrays

You can iterate over nested arrays:

```
[[1, 2], [3, 4]].each { |(a, b)| p "a: #{ a }", "b: #{ b }" }
```

The following syntax is allowed too:

```
[[1, 2], [3, 4]].each { |a, b| "a: #{ a }", "b: #{ b }" }
```

Will produce:

```
"a: 1"
"b: 2"
"a: 3"
"b: 4"
```

Hashes

You can iterate over key-value pairs:

```
{a: 1, b: 2, c: 3}.each { |pair| p "pair: #{ pair }" }
```

Will produce:

```
"pair: [:a, 1]"
"pair: [:b, 2]"
"pair: [:c, 3]"
```

You can iterate over keys and values simultaneously:

```
{a: 1, b: 2, c: 3}.each { |(k, v)| p "k: #{ k }", "v: #{ k }" }
```

Will produce:

```
"k: a"
"v: a"
"k: b"
"v: b"
"k: c"
"v: c"
```

Section 21.4: For iterator

This iterates from 4 to 13 (inclusive).

```
for i in 4..13
    puts "this is #{i}.th number"
end
```

We can also iterate over arrays using for

```
names = ['Siva', 'Charan', 'Naresh', 'Manish']

for name in names
    puts name
end
```

Section 21.5: Iteration with index

Sometimes you want to know the position (**index**) of the current element while iterating over an enumerator. For such purpose, Ruby provides the `with_index` method. It can be applied to all the enumerators. Basically, by adding `with_index` to an enumeration, you can enumerate that enumeration. Index is passed to a block as the second argument.

```
[2,3,4].map.with_index { |e, i| puts "Element of array number #{i} => #{e}" }
#Element of array number 0 => 2
#Element of array number 1 => 3
#Element of array number 2 => 4
#=> [nil, nil, nil]
```

`with_index` has an optional argument – the first index which is `0` by default:

```
[2,3,4].map.with_index(1) { |e, i| puts "Element of array number #{i} => #{e}" }
#Element of array number 1 => 2
#Element of array number 2 => 3
#Element of array number 3 => 4
#=> [nil, nil, nil]
```

There is a specific method `each_with_index`. The only difference between it and `each.with_index` is that you can't pass an argument to that, so the first index is `0` all the time.

```
[2,3,4].each_with_index { |e, i| puts "Element of array number #{i} => #{e}" }
#Element of array number 0 => 2
#Element of array number 1 => 3
#Element of array number 2 => 4
#=> [2, 3, 4]
```

Section 21.6: Map

Returns the changed object, but the original object remains as it was. For example:

```
arr = [1, 2, 3]
arr.map { |i| i + 1 } # => [2, 3, 4]
arr # => [1, 2, 3]
```

`map!` changes the original object:

```
arr = [1, 2, 3]
arr.map! { |i| i + 1 } # => [2, 3, 4]
arr # => [2, 3, 4]
```

Note: you can also use `collect` to do the same thing.

Chapter 22: Exceptions

Section 22.1: Creating a custom exception type

A custom exception is any class that extends `Exception` or a subclass of `Exception`.

In general, you should always extend `StandardError` or a descendant. The `Exception` family are usually for virtual-machine or system errors, rescuing them can prevent a forced interruption from working as expected.

```
# Defines a new custom exception called FileNotFound
class FileNotFound < StandardError
end

def read_file(path)
  File.exist?(path) || raise(FileNotFound, "File #{path} not found")
  File.read(path)
end

read_file("missing.txt")  #=> raises FileNotFound.new("File `missing.txt` not found")
read_file("valid.txt")    #=> reads and returns the content of the file
```

It's common to name exceptions by adding the `Error` suffix at the end:

- `ConnectionError`
- `DontPanicError`

However, when the error is self-explanatory, you don't need to add the `Error` suffix because would be redundant:

- `FileNotFound` vs `FileNotFoundError`
- `DatabaseExploded` vs `DatabaseExplodedError`

Section 22.2: Handling multiple exceptions

You can handle multiple errors in the same `rescue` declaration:

```
begin
  # an execution that may fail
rescue FirstError, SecondError => e
  # do something if a FirstError or SecondError occurs
end
```

You can also add multiple `rescue` declarations:

```
begin
  # an execution that may fail
rescue FirstError => e
  # do something if a FirstError occurs
rescue SecondError => e
  # do something if a SecondError occurs
rescue => e
  # do something if a StandardError occurs
end
```

The order of the `rescue` blocks is relevant: the first match is the one executed. Therefore, if you put `StandardError` as the first condition and all your exceptions inherit from `StandardError`, then the other `rescue` statements will never be executed.

```ruby
begin
  # an execution that may fail
rescue => e
  # this will swallow all the errors
rescue FirstError => e
  # do something if a FirstError occurs
rescue SecondError => e
  # do something if a SecondError occurs
end
```

Some blocks have implicit exception handling like `def`, `class`, and `module`. These blocks allow you to skip the `begin` statement.

```ruby
def foo
    ...
rescue CustomError
    ...
ensure
    ...
end
```

Section 22.3: Handling an exception

Use the `begin`/`rescue` block to catch (rescue) an exception and handle it:

```ruby
begin
  # an execution that may fail
rescue
  # something to execute in case of failure
end
```

A `rescue` clause is analogous to a `catch` block in a curly brace language like C# or Java.

A bare `rescue` like this rescues `StandardError`.

Note: Take care to avoid catching `Exception` instead of the default `StandardError`. The `Exception` class includes `SystemExit` and `NoMemoryError` and other serious exceptions that you usually don't want to catch. Always consider catching `StandardError` (the default) instead.

You can also specify the exception class that should be rescued:

```ruby
begin
  # an excecution that may fail
rescue CustomError
  # something to execute in case of CustomError
  # or descendant
end
```

This rescue clause will not catch any exception that is not a `CustomError`.

You can also store the exception in a specific variable:

```ruby
begin
  # an excecution that may fail
rescue CustomError => error
  # error contains the exception
  puts error.message # provide human-readable details about what went wrong.
```

```
  puts error.backtrace.inspect # return an array of strings that represent the call stack
end
```

If you failed to handle an exception, you can raise it any time in a rescue block.

```
begin
   #here goes your code
rescue => e
    #failed to handle
    raise e
end
```

If you want to retry your `begin` block, call `retry`:

```
begin
   #here goes your code
rescue StandardError => e
   #for some reason you want to retry you code
   retry
end
```

You can be stuck in a loop if you catch an exception in every retry. To avoid this, limit your `retry_count` to a certain number of tries.

```
retry_count = 0
begin
      # an excecution that may fail
rescue
    if retry_count < 5
        retry_count = retry_count + 1
        retry
    else
        #retry limit exceeds, do something else
    end
```

You can also provide an `else` block or an `ensure` block. An `else` block will be executed when the `begin` block completes without an exception thrown. An `ensure` block will always be executed. An `ensure` block is analogous to a `finally` block in a curly brace language like C# or Java.

```
begin
  # an execution that may fail
rescue
  # something to execute in case of failure
else
  # something to execute in case of success
ensure
  # something to always execute
end
```

If you are inside a `def`, `module` or `class` block, there is no need to use the begin statement.

```
def foo
    ...
rescue
    ...
end
```

Section 22.4: Raising an exception

To raise an exception use `Kernel#raise` passing the exception class and/or message:

```
raise StandardError # raises a StandardError.new
raise StandardError, "An error" # raises a StandardError.new("An error")
```

You can also simply pass an error message. In this case, the message is wrapped into a `RuntimeError`:

```
raise "An error" # raises a RuntimeError.new("An error")
```

Here's an example:

```
def hello(subject)
  raise ArgumentError, "`subject` is missing" if subject.to_s.empty?
  puts "Hello #{subject}"
end

hello # => ArgumentError: `subject` is missing
hello("Simone") # => "Hello Simone"
```

Section 22.5: Adding information to (custom) exceptions

It may be helpful to include additional information with an exception, e.g. for logging purposes or to allow conditional handling when the exception is caught:

```
class CustomError < StandardError
  attr_reader :safe_to_retry

  def initialize(safe_to_retry = false, message = 'Something went wrong')
    @safe_to_retry = safe_to_retry
    super(message)
  end
end
```

Raising the exception:

```
raise CustomError.new(true)
```

Catching the exception and accessing the additional information provided:

```
begin
  # do stuff
rescue CustomError => e
  retry if e.safe_to_retry
end
```

Chapter 23: Enumerators

Parameter	Details
`yield`	Responds to `yield`, which is aliased as `<<`. Yielding to this object implements iteration.

An `Enumerator` is an object that implements iteration in a controlled fashion.

Instead of looping until some condition is satisfied, the object *enumerates* values as needed. Execution of the loop is paused until the next value is requested by the owner of the object.

Enumerators make infinite streams of values possible.

Section 23.1: Custom enumerators

Let's create an `Enumerator` for Fibonacci numbers.

```
fibonacci = Enumerator.new do |yielder|
  a = b = 1
  loop do
    yielder << a
    a, b = b, a + b
  end
end
```

We can now use any **`Enumerable`** method with `fibonacci`:

```
fibonacci.take 10
# => [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Section 23.2: Existing methods

If an iteration method such as `each` is called without a block, an `Enumerator` should be returned.

This can be done using the `enum_for` method:

```
def each
  return enum_for :each unless block_given?

  yield :x
  yield :y
  yield :z
end
```

This enables the programmer to compose **`Enumerable`** operations:

```
each.drop(2).map(&:upcase).first
# => :Z
```

Section 23.3: Rewinding

Use `rewind` to restart the enumerator.

```
ℕ = Enumerator.new do |yielder|
  x = 0
  loop do
```

```
    yielder << x
    x += 1
  end
end

ℕ.next
# => 0

ℕ.next
# => 1

ℕ.next
# => 2

ℕ.rewind

ℕ.next
# => 0
```

Chapter 24: Enumerable in Ruby

Enumberable module, a set of methods are available to do traversing, sorting, searching etc across the collection(Array, Hashes, Set, HashMap).

Section 24.1: Enumerable module

```
1. For Loop:
CountriesName = ["India", "Canada", "America", "Iraq"]
for country in CountriesName
    puts country
end

2. Each Iterator:
Same set of work can be done with each loop which we did with for loop.
CountriesName = ["India", "Canada", "America", "Iraq"]
CountriesName.each do |country|
    puts country
end


Each iterator, iterate over every single element of the array.
each ---------- iterator
do ------------ start of the block
|country| ----- argument passed to the block
puts country----block



3. each_with_index Iterator:
each_with_index iterator provides the element for the current iteration and index of the element in
that specific collection.
CountriesName = ["India", "Canada", "America", "Iraq"]
CountriesName.each_with_index do |country, index|
    puts country + "   " + index.to_s
end


4. each_index Iterator:
Just to know the index at which the element is placed in the collection.
CountriesName = ["India", "Canada", "America", "Iraq"]
CountriesName.each_index do |index|
    puts index
end


5. map:
"map" acts as an iterator and also used to fetch the transformed copy of the array. To fetch the
new set of the array rather than introducing the change in the same specific array.
Let us deal with for loop first:
You have an array arr = [1,2,3,4,5]
You need to produce new set of array.
arr = [1,2,3,4,5]
newArr = []
for x in 0..arr.length-1
   newArr[x] = -arr[x]
end


The above mentioned array can be iterated and can produce new set of the array using map method.
```

```ruby
arr = [1,2,3,4,5]
newArr = arr.map do |x|
    -x
      end

puts arr
[1,2,3,4,5]

puts newArr
[-1, -2, -3, -4, -5]
```

map is returning the modified copy of the current value of the collection. arr has unaltered value.

Difference between each and map:
1. map returned the modified value of the collection.

Let us see the example:

```ruby
arr = [1,2,3,4,5]
newArr = arr.map do |x|
    puts x
    -x
      end

puts newArr
[-1, -2, -3, -4, -5]
```

map method is the iterator and also return the copy of transformed collection.

```ruby
arr = [1,2,3,4,5]
newArr = arr.each do |x|
    puts x
    -x
      end

puts newArr
[1,2,3,4,5]
```

each block will throw the array because this is just the iterator.
Each iteration, does not actually alter each element in the iteration.

6. map!
map with bang changes the orginal collection and returned the modified collection not the copy of the modified collection.

```ruby
arr = [1,2,3,4,5]
arr.map! do |x|
    puts x
    -x
end
puts  arr
[-1, -2, -3, -4, -5]
```

7. Combining map and each_with_index
Here each_with_index will iterator over the collection and map will return the modified copy of the collection.

```ruby
CountriesName = ["India", "Canada", "America", "Iraq"]
newArray =
CountriesName.each_with_index.map do |value, index|
    puts "Value is #{value} and the index is #{index}"
```

```
    "Value is #{value} and the index is #{index}"
end

newArray =
CountriesName.each_with_index.map do |value, index|
    if ((index%2).eql?0)
        puts "Value is #{value} and the index is #{index}"
        "Value is #{value} and the index is #{index}"
    end
end

puts newArray
 ["Value is India and the index is 0", nil, "Value is America and the index is 2", nil]


8. select
MixedArray = [1, "India", 2, "Canada", "America", 4]
MixedArray.select do |value|
    (value.class).eql?Integer
end

select method fetches the result based on satifying certain condition.


9. inject methods
inject method reduces the collection to a certain final value.
Let us say you want to find out the sum of the collection.
With for loop how would it work
arr = [1,2,3,4,5]
sum = 0
for x in 0..arr.length-1
   sum = sum + arr[0]
end
puts sum
15

So above mentioned sum can be reduce by single method
arr = [1,2,3,4,5]
arr.inject(0) do |sum, x|
    puts x
    sum = sum + x
end
inject(0) - passing initial value sum = 0
If used inject with no argument sum = arr[0]
sum - After each iteration, total is equal to the return value at the end of the block.
x  - refers to the current iteration element

inject method is also an iterator.
```

Summary: Best way to transform the collection is to make use of Enumerable module to compact the clunky code.

Chapter 25: Classes

Section 25.1: Constructor

A class can have only one constructor, that is a method called `initialize`. The method is automatically invoked when a new instance of the class is created.

```
class Customer
  def initialize(name)
     @name = name.capitalize
  end
end

sarah = Customer.new('sarah')
sarah.name #=> 'Sarah'
```

Section 25.2: Creating a class

You can define a new class using the `class` keyword.

```
class MyClass
end
```

Once defined, you can create a new instance using the `.new` method

```
somevar = MyClass.new
# => #<MyClass:0x007fe2b8aa4a18>
```

Section 25.3: Access Levels

Ruby has three access levels. They are `public`, `private` and `protected`.

Methods that follow the `private` or `protected` keywords are defined as such. Methods that come before these are implicitly `public` methods.

Public Methods

A public method should describe the behavior of the object being created. These methods can be called from outside the scope of the created object.

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

  def speak
    puts "I'm #{@name} and I'm 2 years old"
  end

  ...
end

new_cat = Cat.new("garfield")
#=> <Cat:0x2321868 @name="garfield">

new_cat.speak
```

```
#=> I'm garfield and I'm 2 years old
```

These methods are public ruby methods, they describe the behavior for initializing a new cat and the behavior of the speak method.

`public` keyword is unnecessary, but can be used to escape `private` or `protected`

```
def MyClass
  def first_public_method
  end

  private

  def private_method
  end

  public

  def second_public_method
  end
end
```

Private Methods

Private methods are not accessible from outside of the object. They are used internally by the object. Using the cat example again:

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

  def speak
    age = calculate_cat_age # here we call the private method
    puts "I'm #{@name} and I'm #{age} years old"
  end

  private
     def calculate_cat_age
       2 * 3 - 4
     end
end

my_cat = Cat.new("Bilbo")
my_cat.speak #=> I'm Bilbo and I'm 2 years old
my_cat.calculate_cat_age #=> NoMethodError: private method `calculate_cat_age' called for
#<Cat:0x2321868 @name="Bilbo">
```

As you can see in the example above, the newly created Cat object has access to the `calculate_cat_age` method internally. We assign the variable `age` to the result of running the private `calculate_cat_age` method which prints the name and age of the cat to the console.

When we try and call the `calculate_cat_age` method from outside the `my_cat` object, we receive a **`NoMethodError`** because it's private. Get it?

Protected Methods

Protected methods are very similar to private methods. They cannot be accessed outside the instance of object in

the same way private methods can't be. However, using the `self` ruby method, protected methods can be called within the context of an object of the same type.

```
class Cat
  def initialize(name, age)
    @name = name
    @age = age
  end

  def speak
    puts "I'm #{@name} and I'm #{@age} years old"
  end

  # this == method allows us to compare two objects own ages.
  # if both Cat's have the same age they will be considered equal.
  def ==(other)
     self.own_age == other.own_age
  end

  protected
     def own_age
        self.age
     end
end

cat1 = Cat.new("ricky", 2)
=> #<Cat:0x007fe2b8aa4a18 @name="ricky", @age=2>

cat2 = Cat.new("lucy", 4)
=> #<Cat:0x008gfb7aa6v67 @name="lucy", @age=4>

cat3 = Cat.new("felix", 2)
=> #<Cat:0x009frbaa8V76 @name="felix", @age=2>
```

You can see we've added an age parameter to the cat class and created three new cat objects with the name and age. We are going to call the `own_age` protected method to compare the age's of our cat objects.

```
cat1 == cat2
=> false

cat1 == cat3
=> true
```

Look at that, we were able to retrieve cat1's age using the `self.own_age` protected method and compare it against cat2's age by calling `cat2.own_age` inside of cat1.

Section 25.4: Class Methods types

Classes have 3 types of methods: instance, singleton and class methods.

Instance Methods

These are methods that can be called from an `instance` of the class.

```
class Thing
  def somemethod
    puts "something"
  end
end
```

```ruby
foo = Thing.new # create an instance of the class
foo.somemethod # => something
```

Class Method

These are static methods, i.e, they can be invoked on the class, and not on an instantiation of that class.

```ruby
class Thing
  def Thing.hello(name)
    puts "Hello, #{name}!"
  end
end
```

It is equivalent to use `self` in place of the class name. The following code is equivalent to the code above:

```ruby
class Thing
  def self.hello(name)
    puts "Hello, #{name}!"
  end
end
```

Invoke the method by writing

```ruby
Thing.hello("John Doe")  # prints: "Hello, John Doe!"
```

Singleton Methods

These are only available to specific instances of the class, but not to all.

```ruby
# create an empty class
class Thing
end

# two instances of the class
thing1 = Thing.new
thing2 = Thing.new

# create a singleton method
def thing1.makestuff
  puts "I belong to thing one"
end

thing1.makestuff # => prints: I belong to thing one
thing2.makestuff # NoMethodError: undefined method `makestuff' for #<Thing>
```

Both the `singleton` and `class` methods are called `eigenclasses`. Basically, what ruby does is to create an anonymous class that holds such methods so that it won't interfere with the instances that are created.

Another way of doing this is by the `class <<` constructor. For example:

```ruby
# a class method (same as the above example)
class Thing
  class << self # the anonymous class
    def hello(name)
      puts "Hello, #{name}!"
    end
  end
end

Thing.hello("sarah") # => Hello, sarah!
```

```ruby
# singleton method

class Thing
end

thing1 = Thing.new

class << thing1
  def makestuff
    puts "I belong to thing one"
  end
end

thing1.makestuff # => prints: "I belong to thing one"
```

Section 25.5: Accessing instance variables with getters and setters

We have three methods:

1. `attr_reader`: used to allow reading the variable outside the class.
2. `attr_writer`: used to allow modifying the variable outside the class.
3. `attr_accessor`: combines both methods.

```ruby
class Cat
  attr_reader :age # you can read the age but you can never change it
  attr_writer :name # you can change name but you are not allowed to read
  attr_accessor :breed # you can both change the breed and read it

  def initialize(name, breed)
    @name = name
    @breed = breed
    @age = 2
  end
  def speak
    puts "I'm #{@name} and I am a #{@breed} cat"
  end
end

my_cat = Cat.new("Banjo", "birman")
# reading values:

my_cat.age  #=> 2
my_cat.breed #=> "birman"
my_cat.name #=> Error

# changing values

my_cat.age = 3 #=> Error
my_cat.breed = "sphynx"
my_cat.name = "Bilbo"

my_cat.speak #=> I'm Bilbo and I am a sphynx cat
```

Note that the parameters are symbols. this works by creating a method.

```ruby
class Cat
  attr_accessor :breed
end
```

Is basically the same as:

```
class Cat
  def breed
    @breed
  end
  def breed= value
    @breed = value
  end
end
```

Section 25.6: New, allocate, and initialize

In many languages, new instances of a class are created using a special `new` keyword. In Ruby, `new` is also used to create instances of a class, but it isn't a keyword; instead, it's a static/class method, no different from any other static/class method. The definition is roughly this:

```
class MyClass
   def self.new(*args)
     obj = allocate
     obj.initialize(*args) # oversimplied; initialize is actually private
     obj
   end
end
```

`allocate` performs the real 'magic' of creating an uninitialized instance of the class

Note also that the return value of `initialize` is discarded, and obj is returned instead. This makes it immediately clear why you can code your initialize method without worrying about returning `self` at the end.

The 'normal' `new` method that all classes get from `Class` works as above, but it's possible to redefine it however you like, or to define alternatives that work differently. For example:

```
class MyClass
  def self.extraNew(*args)
    obj = allocate
    obj.pre_initialize(:foo)
    obj.initialize(*args)
    obj.post_initialize(:bar)
    obj
  end
end
```

Section 25.7: Dynamic class creation

Classes can be created dynamically through the use of `Class.new`.

```
# create a new class dynamically
MyClass = Class.new

# instantiate an object of type MyClass
my_class = MyClass.new
```

In the above example, a new class is created and assigned to the constant `MyClass`. This class can be instantiated and used just like any other class.

The `Class.new` method accepts a `Class` which will become the superclass of the dynamically created class.

```ruby
# dynamically create a class that subclasses another
Staffy = Class.new(Dog)

# instantiate an object of type Staffy
lucky = Staffy.new
lucky.is_a?(Staffy) # true
lucky.is_a?(Dog)    # true
```

The `Class.new` method also accepts a block. The context of the block is the newly created class. This allows methods to be defined.

```ruby
Duck =
  Class.new do
    def quack
      'Quack!!'
    end
  end

# instantiate an object of type Duck
duck = Duck.new
duck.quack # 'Quack!!'
```

Section 25.8: Class and instance variables

There are several special variable types that a class can use for more easily sharing data.

Instance variables, preceded by `@`. They are useful if you want to use the same variable in different methods.

```ruby
class Person
  def initialize(name, age)
    my_age = age # local variable, will be destroyed at end of constructor
    @name = name # instance variable, is only destroyed when the object is
  end

  def some_method
    puts "My name is #{@name}." # we can use @name with no problem
  end

  def another_method
    puts "My age is #{my_age}." # this will not work!
  end
end

mhmd = Person.new("Mark", 23)

mhmd.some_method #=> My name is Mark.
mhmd.another_method #=> throws an error
```

Class variable, preceded by `@@`. They contain the same values across all instances of a class.

```ruby
class Person
  @@persons_created = 0 # class variable, available to all objects of this class
  def initialize(name)
    @name = name

    # modification of class variable persists across all objects of this class
    @@persons_created += 1
  end
```

```ruby
  def how_many_persons
    puts "persons created so far: #{@@persons_created}"
  end
end

mark = Person.new("Mark")
mark.how_many_persons #=> persons created so far: 1
helen = Person.new("Helen")

mark.how_many_persons #=> persons created so far: 2
helen.how_many_persons #=> persons created so far: 2
# you could either ask mark or helen
```

Global Variables, preceded by $. These are available anywhere to the program, so make sure to use them wisely.

```ruby
$total_animals = 0

class Cat
  def initialize
    $total_animals += 1
  end
end

class Dog
  def initialize
    $total_animals += 1
  end
end

bob = Cat.new()
puts $total_animals #=> 1
fred = Dog.new()
puts $total_animals #=> 2
```

Chapter 26: Inheritance

Section 26.1: Subclasses

Inheritance allows classes to define specific behaviour based on an existing class.

```
class Animal
  def say_hello
    'Meep!'
  end

  def eat
    'Yumm!'
  end
end

class Dog < Animal
  def say_hello
    'Woof!'
  end
end

spot = Dog.new
spot.say_hello # 'Woof!'
spot.eat       # 'Yumm!'
```

In this example:

- `Dog` Inherits from `Animal`, making it a *Subclass*.
- `Dog` gains both the `say_hello` and `eat` methods from `Animal`.
- `Dog` overrides the `say_hello` method with different functionality.

Section 26.2: What is inherited?

Methods are inherited

```
class A
  def boo; p 'boo' end
end

class B < A; end

b = B.new
b.boo # => 'boo'
```

Class methods are inherited

```
class A
  def self.boo; p 'boo' end
end

class B < A; end

p B.boo # => 'boo'
```

Constants are inherited

```ruby
class A
  WOO = 1
end

class B < A; end

p B::WOO # => 1
```

But beware, they can be overridden:

```ruby
class B
  WOO = WOO + 1
end

p B::WOO # => 2
```

Instance variables are inherited:

```ruby
class A
  attr_accessor :ho
  def initialize
    @ho = 'haha'
  end
end

class B < A; end

b = B.new
p b.ho # => 'haha'
```

Beware, if you override the methods that initialize instance variables without calling `super`, they will be nil. Continuing from above:

```ruby
class C < A
  def initialize; end
 end

c = C.new
p c.ho    # => nil
```

Class instance variables are not inherited:

```ruby
class A
    @foo = 'foo'
    class << self
        attr_accessor :foo
    end
end

class B < A; end

p B.foo # => nil

# The accessor is inherited, since it is a class method
#
B.foo = 'fob' # possible
```

Class variables aren't really inherited

They are shared between the base class and all subclasses as 1 variable:

```
class A
    @@foo = 0
    def initialize
        @@foo  += 1
        p @@foo
    end
end

class B < A;end

a = A.new # => 1
b = B.new # => 2
```

So continuing from above:

```
class C < A
  def initialize
    @@foo = -10
    p @@foo
  end
end

a = C.new # => -10
b = B.new # => -9
```

Section 26.3: Multiple Inheritance

Multiple inheritance is a feature that allows one class to inherit from multiple classes(i.e., more than one parent). Ruby does not support multiple inheritance. It only supports single-inheritance (i.e. class can have only one parent), but you can use *composition* to build more complex classes using Modules.

Section 26.4: Mixins

Mixins are a beautiful way to achieve something similar to multiple inheritance. It allows us to inherit or rather include methods defined in a module into a class. These methods can be included as either instance or class methods. The below example depicts this design.

```
module SampleModule

  def self.included(base)
    base.extend ClassMethods
  end

  module ClassMethods

    def method_static
      puts "This is a static method"
    end

  end

  def insta_method
    puts "This is an instance method"
  end

end
```

```ruby
class SampleClass
  include SampleModule
end

sc = SampleClass.new

sc.insta_method

prints "This is an instance method"

sc.class.method_static

prints "This is a static method"
```

Section 26.5: Refactoring existing classes to use Inheritance

Let's say we have two classes, `Cat` and `Dog`.

```ruby
class Cat
  def eat
    die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  def sound
    puts "Meow"
  end
end

class Dog
  def eat
    die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  def sound
    puts "Woof"
  end
end
```

The `eat` method is exactly the same in these two classes. While this works, it is hard to maintain. The problem will get worse if there are more animals with the same `eat` method. Inheritance can solve this problem.

```ruby
class Animal
  def eat
    die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  # No sound method
end

class Cat < Animal
  def sound
    puts "Meow"
  end
end

class Dog < Animal
  def sound
```

```
    puts "Woof"
  end
end
```

We have created a new class, `Animal`, and moved our `eat` method to that class. Then, we made `Cat` and `Dog` inherit from this new common superclass. This removes the need for repeating code

Chapter 27: method_missing

Parameter	Details
method	The name of the method that has been called (in the above example this is `:say_moo`, note that this is a symbol.
*args	The arguments passed in to this method. Can be any number, or none
&block	The block of the method called, this can either be a `do` block, or a `{ }` enclosed block

Section 27.1: Catching calls to an undefined method

```
class Animal
  def method_missing(method, *args, &block)
    "Cannot call #{method} on Animal"
  end
end

=> Animal.new.say_moo
> "Cannot call say_moo on Animal"
```

Section 27.2: Use with block

```
class Animal
  def method_missing(method, *args, &block)
    if method.to_s == 'say'
      block.call
    else
      super
    end
  end
end

 => Animal.new.say{ 'moo' }
 => "moo"
```

Section 27.3: Use with parameter

```
class Animal
  def method_missing(method, *args, &block)
    say, speak = method.to_s.split("_")
    if say == "say" && speak
      return speak.upcase if args.first == "shout"
      speak
    else
      super
    end
  end
end

=> Animal.new.say_moo
=> "moo"
=> Animal.new.say_moo("shout")
=> "MOO"
```

Section 27.4: Using the missing method

```
class Animal
```

```ruby
  def method_missing(method, *args, &block)
    say, speak = method.to_s.split("_")
    if say == "say"
      speak
    else
      super
    end
  end
end

=> a = Animal.new
=> a.say_moo
=> "moo"
=> a.shout_moo
=> NoMethodError: undefined method `shout_moo'
```

Chapter 28: Regular Expressions and Regex Based Operations

Section 28.1: =~ operator

```
if /hay/ =~ 'haystack'
  puts "There is hay in the word haystack"
end
```

Note: The order **is significant**. Though `'haystack' =~ /hay/` is in most cases an equivalent, side effects might differ:

- Strings captured from named capture groups are assigned to local variables only when `Regexp#=~` is called (`regexp =~ str`);
- Since the right operand might be is an arbitrary object, for `regexp =~ str` there will be called either `Regexp#=~` or `String#=~`.

Note that this does not return a true/false value, it instead returns either the index of the match if found, or nil if not found. Because all integers in ruby are truthy (including 0) and nil is falsy, this works. If you want a boolean value, use

#===

as shown in another example.

Section 28.2: Regular Expressions in Case Statements

You can test if a string matches several regular expressions using a switch statement.

Example

```
case "Ruby is #1!"
when /\APython/
    puts "Boooo."
when /\ARuby/
    puts "You are right."
else
    puts "Sorry, I didn't understand that."
end
```

This works because case statements are checked for equality using the `===` operator, not the `==` operator. When a regex is on the left hand side of a comparison using `===`, it will test a string to see if it matches.

Section 28.3: Groups, named and otherwise

Ruby extends the standard group syntax `(...)` with a named group, `(?<name>...)`. This allows for extraction by name instead of having to count how many groups you have.

```
name_reg = /h(i|ello), my name is (?<name>.*)/i #i means case insensitive

name_input = "Hi, my name is Zaphod Beeblebrox"

match_data = name_reg.match(name_input) #returns either a MatchData object or nil
match_data = name_input.match(name_reg) #works either way
```

```ruby
if match_data.nil? #Always check for nil! Common error.
  puts "No match"
else
  match[0] #=> "Hi, my name is Zaphod Beeblebrox"
  match[1] #=> "i" #the first group, (i|ello)
  match[2] #=> "Zaphod Beeblebrox"
  #Because it was a named group, we can get it by name
  match[:name]  #=> "Zaphod Beeblebrox"
  match["name"] #=> "Zaphod Beeblebrox"
  puts "Hello #{match[:name]}!"
end
```

The index of the match is counted based on the order of the left parentheses (with the entire regex being the first group at index 0)

```ruby
reg = /(((a)b)c)(d)/
match = reg.match 'abcd'
match[0] #=> "abcd"
match[1] #=> "abc"
match[2] #=> "ab"
match[3] #=> "a"
match[4] #=> "d"
```

Section 28.4: Quantifiers

Quantifiers allows to specify count of repeated strings.

- Zero or one:

```ruby
/a?/
```

- Zero or many:

```ruby
/a*/
```

- One or many:

```ruby
/a+/
```

- Exact number:

```ruby
/a{2,4}/ # Two, three or four
/a{2,}/  # Two or more
/a{,4}/  # Less than four (including zero)
```

By default, quantifiers are greedy, which means they take as many characters as they can while still making a match. Normally this is not noticeable:

```ruby
/(?<site>.*) Stack Exchange/ =~ 'Motor Vehicle Maintenance & Repair Stack Exchange'
```

The named capture group `site` will be set to "Motor Vehicle Maintenance & Repair' as expected. But if 'Stack Exchange' is an optional part of the string (because it could be 'Stack Overflow' instead), the naive solution will not work as expected:

```
/(?<site>.*)( Stack Exchange)?/
```

This version will still match, but the named capture will include 'Stack Exchange' since `*` greedily eats those characters. The solution is to add another question mark to make the `*` lazy:

```
/(?<site>.*?)( Stack Exchange)?/
```

Appending `?` to any quantifier will make it lazy.

Section 28.5: Common quick usage

Regular expressions are often used in methods as parameters to check if other strings are present or to search and/or replace strings.

You'll often see the following:

```
string = "My not so long string"
string[/so/] # gives so
string[/present/] # gives nil
string[/present/].nil? # gives true
```

So you can simply use this as a check if a string contains a substring

```
puts "found" if string[/so/]
```

More advanced but still short and quick: search for a specific group by using the second parameter, 2 is the second in this example because numbering starts at 1 and not 0, a group is what is enclosed in parentheses.

```
string[/(n.t).+(l.ng)/, 2] # gives long
```

Also often used: search and replace with `sub` or `gsub`, `\1` gives the first found group, `\2` the second:

```
string.gsub(/(n.t).+(l.ng)/, '\1 very \2') # My not very long string
```

The last result is remembered and can be used on the following lines

```
$2 # gives long
```

Section 28.6: match? - Boolean Result

Returns `true` or `false`, which indicates whether the regexp is matched or not without updating `$~` and other related variables. If the second parameter is present, it specifies the position in the string to begin the search.

```
/R.../.match?("Ruby")    #=> true
/R.../.match?("Ruby", 1) #=> false
/P.../.match?("Ruby")    #=> false
```

Ruby 2.4+

Section 28.7: Defining a Regexp

A Regexp can be created in three different ways in Ruby.

- using slashes: `/ /`
- using `%r{}`
- using `Regex.new`

```
#The following forms are equivalent
regexp_slash = /hello/
regexp_bracket = %r{hello}
regexp_new = Regexp.new('hello')

string_to_match = "hello world!"

#All of these will return a truthy value
string_to_match =~ regexp_slash    # => 0
string_to_match =~ regexp_bracket  # => 0
string_to_match =~ regexp_new      # => 0
```

Section 28.8: Character classes

Describes ranges of symbols

You can enumerate symbols explicitly

```
/[abc]/ # 'a' or 'b' or 'c'
```

Or use ranges

```
/[a-z]/ # from 'a' to 'z'
```

It is possible to combine ranges and single symbols

```
/[a-cz]/ # 'a' or 'b' or 'c' or 'z'
```

Leading dash (-) is treated as character

```
/[-a-c]/ # '-' or 'a' or 'b' or 'c'
```

Classes can be negative when preceding symbols with `^`

```
/[^a-c]/ # Not 'a', 'b' or 'c'
```

There are some shortcuts for widespread classes and special characters, plus line endings

```
^  # Start of line
$  # End of line
\A # Start of string
\Z # End of string, excluding any new line at the end of string
\z # End of string
.  # Any single character
\s # Any whitespace character
\S # Any non-whitespace character
\d # Any digit
\D # Any non-digit
\w # Any word character (letter, number, underscore)
\W # Any non-word character
```

```
\b # Any word boundary
```

`\n` will be understood simply as new line

To escape any reserved character, such as `/` or `[ ]` and others use backslash (left slash)

```
\\ # => \
\[\] # => []
```

Chapter 29: File and I/O Operations

Flag	Meaning
"r"	Read-only, starts at beginning of file (default mode).
"r+"	Read-write, starts at beginning of file.
"w"	Write-only, truncates existing file to zero length or creates a new file for writing.
"w+"	Read-write, truncates existing file to zero length or creates a new file for reading and writing.
"a"	Write-only, starts at end of file if file exists, otherwise creates a new file for writing.
"a+"	Read-write, starts at end of file if file exists, otherwise creates a new file for reading and writing.
"b"	Binary file mode. Suppresses EOL <-> CRLF conversion on Windows. And sets external encoding to ASCII-8BIT unless explicitly specified. (This flag may only appear in conjunction with the above flags. For example, `File.new("test.txt", "rb")` would open `test.txt` in `read-only` mode as a `binary` file.)
"t"	Text file mode. (This flag may only appear in conjunction with the above flags. For example, `File.new("test.txt", "wt")` would open `test.txt` in `write-only` mode as a `text` file.)

Section 29.1: Writing a string to a file

A string can be written to a file with an instance of the `File` class.

```
file = File.new('tmp.txt', 'w')
file.write("NaNaNaNa\n")
file.write('Batman!\n')
file.close
```

The `File` class also offers a shorthand for the `new` and `close` operations with the `open` method.

```
File.open('tmp.txt', 'w') do |f|
  f.write("NaNaNaNa\n")
  f.write('Batman!\n')
end
```

For simple write operations, a string can be also written directly to a file with `File.write`. **Note that this will overwrite the file by default.**

```
File.write('tmp.txt', "NaNaNaNa\n" * 4 + 'Batman!\n')
```

To specify a different mode on `File.write`, pass it as the value of a key called `mode` in a hash as another parameter.

```
File.write('tmp.txt', "NaNaNaNa\n" * 4 + 'Batman!\n', { mode: 'a'})
```

Section 29.2: Reading from STDIN

```
# Get two numbers from STDIN, separated by a newline, and output the result
number1 = gets
number2 = gets
puts number1.to_i + number2.to_i
## run with: $ ruby a_plus_b.rb
## or:       $ echo -e "1\n2" | ruby a_plus_b.rb
```

Section 29.3: Reading from arguments with ARGV

```
number1 = ARGV[0]
number2 = ARGV[1]
puts number1.to_i + number2.to_i
```

```
## run with: $ ruby a_plus_b.rb 1 2
```

Section 29.4: Open and closing a file

Manually open and close a file.

```
# Using new method
f = File.new("test.txt", "r") # reading
f = File.new("test.txt", "w") # writing
f = File.new("test.txt", "a") # appending

# Using open method
f = open("test.txt", "r")

# Remember to close files
f.close
```

Automatically close a file using a block.

```
f = File.open("test.txt", "r") do |f|
  # do something with file f
  puts f.read # for example, read it
end
```

Section 29.5: get a single char of input

Unlike `gets.chomp` this will not wait for a newline.

First part of the stdlib must be included

```
require 'io/console'
```

Then a helper method can be written:

```
def get_char
  input = STDIN.getch
  control_c_code = "\u0003"
  exit(1) if input == control_c_code
  input
end
```

Its' imporant to exit if `control+c` is pressed.

Chapter 30: Ruby Access Modifiers

Access control(scope) to various methods, data members, initialize methods.

Section 30.1: Instance Variables and Class Variables

Let's first brush up with what are the **Instance Variables:** They behave more like properties for an object. They are initialized on an object creation. Instance variables are accessible through instance methods. Per Object has per instance variables. Instance Variables are not shared between objects.

Sequence class has @from, @to and @by as the instance variables.

```
class Sequence
    include Enumerable

    def initialize(from, to, by)
        @from = from
        @to = to
        @by = by
    end

    def each
        x = @from
        while x < @to
            yield x
            x = x + @by
        end
    end

    def *(factor)
        Sequence.new(@from*factor, @to*factor, @by*factor)
    end

    def +(offset)
        Sequence.new(@from+offset, @to+offset, @by+offset)
    end
end

object = Sequence.new(1,10,2)
object.each do |x|
    puts x
end

Output:
1
3
5
7
9

object1 = Sequence.new(1,10,3)
object1.each do |x|
    puts x
end

Output:
1
4
```

```
7
```

Class Variables Treat class variable same as static variables of java, which are shared among the various objects of that class. Class Variables are stored in heap memory.

```
class Sequence
    include Enumerable
    @@count = 0
    def initialize(from, to, by)
        @from = from
        @to = to
        @by = by
        @@count = @@count + 1
    end

    def each
        x = @from
        while x < @to
            yield x
            x = x + @by
        end
    end

    def *(factor)
        Sequence.new(@from*factor, @to*factor, @by*factor)
    end

    def +(offset)
        Sequence.new(@from+offset, @to+offset, @by+offset)
    end

    def getCount
        @@count
    end
end

object = Sequence.new(1,10,2)
object.each do |x|
    puts x
end

Output:
1
3
5
7
9

object1 = Sequence.new(1,10,3)
object1.each do |x|
    puts x
end

Output:
1
4
7

puts object1.getCount
Output: 2
```

Shared among object and object1.

Comparing the instance and class variables of Ruby against Java:

```
Class Sequence{
    int from, to, by;
    Sequence(from, to, by){// constructor method of Java is equivalent to initialize method of ruby
        this.from = from;// this.from of java is equivalent to @from indicating currentObject.from
        this.to = to;
        this.by = by;
    }
    public void each(){
        int x = this.from;//objects attributes are accessible in the context of the object.
        while x > this.to
            x = x + this.by
    }
}
```

Section 30.2: Access Controls

Comparison of access controls of Java against Ruby: If method is declared private in Java, it can only be accessed by other methods within the same class. If a method is declared protected it can be accessed by other classes which exist within the same package as well as by subclasses of the class in a different package. When a method is public it is visible to everyone. In Java, access control visibility concept depends on where these classes lie's in the inheritance/package hierarchy.

Whereas in Ruby, the inheritance hierarchy or the package/module don't fit. It's all about which object is the receiver of a method.

For a private method in Ruby, it can never be called with an explicit receiver. We can (only) call the private method with an implicit receiver.

This also means we can call a private method from within a class it is declared in as well as all subclasses of this class.

```
class Test1
  def main_method
    method_private
  end

  private
  def method_private
    puts "Inside methodPrivate for #{self.class}"
  end
end

class Test2 < Test1
  def main_method
    method_private
  end
end

Test1.new.main_method
Test2.new.main_method

Inside methodPrivate for Test1
Inside methodPrivate for Test2
```

```ruby
class Test3 < Test1
  def main_method
    self.method_private #We were trying to call a private method with an explicit receiver and if
called in the same class with self would fail.
  end
end

Test1.new.main_method
This will throw NoMethodError

You can never call the private method from outside the class hierarchy where it was defined.
```

Protected method can be called with an implicit receiver, as like private. In addition protected method can also be called by an explicit receiver (only) if the receiver is "self" or "an object of the same class".

```ruby
class Test1
  def main_method
    method_protected
  end

  protected
  def method_protected
    puts "InSide method_protected for #{self.class}"
  end
end

class Test2 < Test1
  def main_method
    method_protected # called by implicit receiver
  end
end

class Test3 < Test1
  def main_method
    self.method_protected # called by explicit receiver "an object of the same class"
  end
end


InSide method_protected for Test1
InSide method_protected for Test2
InSide method_protected for Test3


class Test4 < Test1
  def main_method
    Test2.new.method_protected # "Test2.new is the same type of object as self"
  end
end

Test4.new.main_method

class Test5
  def main_method
    Test2.new.method_protected
  end
end

Test5.new.main_method
This would fail as object Test5 is not subclass of Test1
```

Consider Public methods with maximum visibility

Summary

1. **Public:** Public methods have maximum visibility
2. **Protected: Protected method** can be called with an implicit receiver, as like private. In addition protected method can also be called by an explicit receiver (only) if the receiver is "self" or "an object of the same class".
3. **Private: For a private method in Ruby**, it can never be called with an explicit receiver. We can (only) call the private method with an implicit receiver. This also means we can call a private method from within a class it is declared in as well as all subclasses of this class.

Chapter 31: Design Patterns and Idioms in Ruby

Section 31.1: Decorator Pattern

Decorator pattern adds behavior to objects without affecting other objects of the same class. The decorator pattern is a useful alternative to creating sub-classes.

Create a module for each decorator. This approach is more flexible than inheritance because you can mix and match responsibilities in more combinations. Additionally, because the transparency allows decorators to be nested recursively, it allows for an unlimited number of responsibilities.

Assume the Pizza class has a cost method that returns 300:

```
class Pizza
  def cost
    300
  end
end
```

Represent pizza with an added layer of cheese burst and the cost goes up by 50. The simplest approach is to create a `PizzaWithCheese` subclass that returns 350 in the cost method.

```
class PizzaWithCheese < Pizza
  def cost
    350
  end
end
```

Next, we need to represent a large pizza that adds 100 to the cost of a normal pizza. We can represent this using a LargePizza subclass of Pizza.

```
class LargePizza < Pizza
  def cost
    400
  end
end
```

We could also have an ExtraLargePizza which adds a further cost of 15 to our LargePizza. If we were to consider that these pizza types could be served with cheese, we would need to add LargePizzaWithChese and ExtraLargePizzaWithCheese subclasses.we end up with a total of 6 classes.

To simplify the approach, use modules to dynamically add behavior to Pizza class:

Module + extend + super decorator:->

```
class Pizza
  def cost
    300
  end
end

module CheesePizza
  def cost
    super + 50
```

```
  end
end

module LargePizza
  def cost
    super + 100
  end
end

pizza = Pizza.new          #=> cost = 300
pizza.extend(CheesePizza) #=> cost = 350
pizza.extend(LargePizza)  #=> cost = 450
pizza.cost                #=> cost = 450
```

Section 31.2: Observer

The observer pattern is a software design pattern in which an object (called `subject`) maintains a list of its dependents (called `observers`), and notifies them automatically of any state changes, usually by calling one of their methods.

Ruby provides a simple mechanism to implement the Observer design pattern. The module `Observable` provides the logic to notify the subscriber of any changes in the Observable object.

For this to work, the observable has to assert it has changed and notify the observers.

Objects observing have to implement an `update()` method, which will be the callback for the Observer.

Let's implement a small chat, where users can subscribe to users and when one of them write something, the subscribers get notified.

```
require "observer"

class Moderator
  include Observable

  def initialize(name)
    @name = name
  end

  def write
    message = "Computer says: No"
    changed
    notify_observers(message)
  end
end

class Warner
  def initialize(moderator, limit)
    @limit = limit
    moderator.add_observer(self)
  end
end

class Subscriber < Warner
  def update(message)
    puts "#{message}"
  end
end
```

```
moderator = Moderator.new("Rupert")
Subscriber.new(moderator, 1)
moderator.write
moderator.write
```

Producing the following output:

```
# Computer says: No
# Computer says: No
```

We've triggered the method `write` at the Moderator class twice, notifying its subscribers, in this case just one.

The more subscribers we add the more the changes will propagate.

Section 31.3: Singleton

Ruby Standard Library has a Singleton module which implements the Singleton pattern. The first step in creating a Singleton class is to require and include the `Singleton` module in a class:

```
require 'singleton'

class Logger
  include Singleton
end
```

If you try to instantiate this class as you normally would a regular class, a `NoMethodError` exception is raised. The constructor is made private to prevent other instances from being accidentally created:

```
Logger.new

#=> NoMethodError: private method `new' called for AppConfig:Class
```

To access the instance of this class, we need to use the `instance()`:

```
first, second = Logger.instance, Logger.instance
first == second

#=> true
```

Logger example

```
require 'singleton'


class Logger
  include Singleton

  def initialize
    @log = File.open("log.txt", "a")
  end

  def log(msg)
    @log.puts(msg)
  end
end
```

In order to use `Logger` object:

```
Logger.instance.log('message 2')
```

Without Singleton include

The above singleton implementations can also be done without the inclusion of the Singleton module. This can be achieved with the following:

```
class Logger
  def self.instance
    @instance ||= new
  end
end
```

which is a shorthand notation for the following:

```
class Logger
  def self.instance
    @instance = @instance || Logger.new
  end
end
```

However, keep in mind that the Singleton module is tested and optimized, therefore being the better option to implement your singleton with.

Section 31.4: Proxy

Proxy object is often used to ensure guarded access to another object, which internal business logic we don't want to pollute with safety requirements.

Suppose we'd like to guarantee that only user of specific permissions can access resource.

Proxy definition: (it ensure that only users which actually can see reservations will be able to consumer reservation_service)

```
class Proxy
  def initialize(current_user, reservation_service)
    @current_user = current_user
    @reservation_service = reservation_service
  end

  def highest_total_price_reservations(date_from, date_to, reservations_count)
    if @current_user.can_see_reservations?
      @reservation_service.highest_total_price_reservations(
        date_from,
        date_to,
        reservations_count
      )
    else
      []
    end
  end
end
```

Models and ReservationService:

```
class Reservation
  attr_reader :total_price, :date
```

```ruby
  def initialize(date, total_price)
    @date = date
    @total_price = total_price
  end
end

class ReservationService
  def highest_total_price_reservations(date_from, date_to, reservations_count)
    # normally it would be read from database/external service
    reservations = [
      Reservation.new(Date.new(2014, 5, 15), 100),
      Reservation.new(Date.new(2017, 5, 15), 10),
      Reservation.new(Date.new(2017, 1, 15), 50)
    ]

    filtered_reservations = reservations.select do |reservation|
      reservation.date.between?(date_from, date_to)
    end

    filtered_reservations.take(reservations_count)
  end
end

class User
  attr_reader :name

  def initialize(can_see_reservations, name)
    @can_see_reservations = can_see_reservations
    @name = name
  end

  def can_see_reservations?
    @can_see_reservations
  end
end
```

Consumer service:

```ruby
class StatsService
  def initialize(reservation_service)
    @reservation_service = reservation_service
  end

  def year_top_100_reservations_average_total_price(year)
    reservations = @reservation_service.highest_total_price_reservations(
      Date.new(year, 1, 1),
      Date.new(year, 12, 31),
      100
    )

    if reservations.length > 0
      sum = reservations.reduce(0) do |memo, reservation|
        memo + reservation.total_price
      end

      sum / reservations.length
    else
      0
    end
  end
end
```

Test:

```
def test(user, year)
  reservations_service = Proxy.new(user, ReservationService.new)
  stats_service = StatsService.new(reservations_service)
  average_price = stats_service.year_top_100_reservations_average_total_price(year)
  puts "#{user.name} will see: #{average_price}"
end

test(User.new(true, "John the Admin"), 2017)
test(User.new(false, "Guest"),         2017)
```

BENEFITS

- **we're avoiding any changes in `ReservationService` when access restrictions are changed.**
- **we're not mixing business related data (`date_from`, `date_to`, `reservations_count`) with domain unrelated concepts (user permissions) in service.**
- **Consumer (`StatsService`) is free from permissions related logic as well**

CAVEATS

- Proxy interface is always exactly the same as the object it hides, so that user that consumes service wrapped by proxy wasn't even aware of proxy presence.

Chapter 32: Loading Source Files

Section 32.1: Require files to be loaded only once

The Kernel#require method will load files only once (several calls to `require` will result in the code in that file being evaluated only once). It will search your ruby `$LOAD_PATH` to find the required file if the parameter is not an absolute path. Extensions like `.rb`, `.so`, `.o` or `.dll` are optional. Relative paths will be resolved to the current working directory of the process.

```
require 'awesome_print'
```

The Kernel#require_relative allows you to load files relative to the file in which `require_relative` is called.

```
# will search in directory myproj relative to current source file.
#
require_relative 'myproj/version'
```

Section 32.2: Automatically loading source files

The method *Kernel#autoload* registers filename to be loaded (using `Kernel::require`) the first time that module (which may be a String or a symbol) is accessed.

```
autoload :MyModule, '/usr/local/lib/modules/my_module.rb'
```

The method Kernel#autoload? returns filename to be loaded if name is registered as `autoload`.

```
autoload? :MyModule  #=> '/usr/local/lib/modules/my_module.rb'
```

Section 32.3: Loading optional files

When files are not available, the `require` family will throw a `LoadError`. This is an example which illustrates loading optional modules only if they exist.

```
module TidBits

@@unavailableModules = []

[
      { name: 'CoreExtend', file: 'core_extend/lib/core_extend'  } \
    , { name: 'Fs'        , file: 'fs/lib/fs'                    } \
    , { name: 'Options'   , file: 'options/lib/options'          } \
    , { name: 'Susu'      , file: 'susu/lib/susu'                } \

].each do |lib|

    begin

        require_relative lib[ :file ]

    rescue LoadError

        @@unavailableModules.push lib

    end
```

```
end

end # module TidBits
```

Section 32.4: Loading files repeatedly

The Kernel#load method will evaluate the code in the given file. The search path will be constructed as with `require`. It will re-evaluate that code on every subsequent call unlike `require`. There is no `load_relative`.

```
load `somefile`
```

Section 32.5: Loading several files

You can use any ruby technique to dynamically create a list of files to load. Illustration of globbing for files starting with `test`, loaded in alphabetical order.

```
Dir[ "#{ __dir__ }**/test*.rb" ) ].sort.each do |source|

    require_relative source

end
```

Chapter 33: Thread

Section 33.1: Accessing shared resources

Use a mutex to synchronise access to a variable which is accessed from multiple threads:

```
counter = 0
counter_mutex = Mutex.new

# Start three parallel threads and increment counter
3.times.map do |index|
  Thread.new do
    counter_mutex.synchronize { counter += 1 }
  end
end.each(&:join) # Wait for all threads to finish before killing the process
```

Otherwise, the value of `counter` currently visible to one thread could be changed by another thread.

Example **without** `Mutex` (see e.g. `Thread 0`, where `Before` and `After` differ by more than `1`):

```
2.2.0 :224 > counter = 0; 3.times.map { |i| Thread.new { puts "[Thread #{i}] Before: #{counter}";
counter += 1; puts "[Thread #{i}] After: #{counter}"; } }.each(&:join)
[Thread 2] Before: 0
[Thread 0] Before: 0
[Thread 0] After: 2
[Thread 1] Before: 0
[Thread 1] After: 3
[Thread 2] After: 1
```

Example **with** `Mutex`:

```
2.2.0 :226 > mutex = Mutex.new; counter = 0; 3.times.map { |i| Thread.new { mutex.synchronize {
puts "[Thread #{i}] Before: #{counter}"; counter += 1; puts "[Thread #{i}] After: #{counter}"; } }
}.each(&:join)
[Thread 2] Before: 0
[Thread 2] After: 1
[Thread 1] Before: 1
[Thread 1] After: 2
[Thread 0] Before: 2
[Thread 0] After: 3
```

Section 33.2: Basic Thread Semantics

A new thread separate from the main thread's execution, can be created using `Thread.new`.

```
thr = Thread.new {
  sleep 1 # 1 second sleep of sub thread
  puts "Whats the big deal"
}
```

This will automatically start the execution of the new thread.

To freeze execution of the main Thread, until the new thread stops, use `join`:

```
thr.join #=> ... "Whats the big deal"
```

Note that the Thread may have already finished when you call join, in which case execution will continue normally. If a sub-thread is never joined, and the main thread completes, the sub-thread will not execute any remaining code.

Section 33.3: Terminating a Thread

A thread terminates if it reaches the end of its code block. The best way to terminate a thread early is to convince it to reach the end of its code block. This way, the thread can run cleanup code before dying.

This thread runs a loop while the instance variable continue is true. Set this variable to false, and the thread will die a natural death:

```
require 'thread'

class CounterThread < Thread
  def initialize
    @count = 0
    @continue = true

    super do
      @count += 1 while @continue
      puts "I counted up to #{@count} before I was cruelly stopped."
    end
  end

  def stop
    @continue = false
  end
end

counter = CounterThread.new
sleep 2
counter.stop
```

Section 33.4: How to kill a thread

You call use `Thread.kill` or `Thread.terminate`:

```
thr = Thread.new { ... }
Thread.kill(thr)
```

Chapter 34: Range

Section 34.1: Ranges as Sequences

The most important use of ranges is to express a sequence

Syntax:

```
(begin..end) => this construct will include end value
(begin...end) => this construct will exclude end value
```

or

```
Range.new(begin,end,exclude_end) => exclude_end is by default false
```

Most important `end` value must be greater the `begin`, otherwise it will return nothing.

Examples:

```
(10..1).to_a            #=> []
(1...3)                 #=> [1, 2]
(-6..-1).to_a           #=> [-6, -5, -4, -3, -2, -1]
('a'..'e').to_a         #=> ["a", "b", "c", "d", "e"]
('a'...'e').to_a        #=> ["a", "b", "c", "d"]
Range.new(1,3).to_a     #=> [1, 2, 3]
Range.new(1,3,true).to_a#=> [1, 2]
```

Section 34.2: Iterating over a range

You can easily do something to each element in a range.

```
(1..5).each do |i|
    print i
end
# 12345
```

Section 34.3: Range between dates

```
require 'date'

date1 = Date.parse "01/06/2016"
date2 = Date.parse "05/06/2016"

p "Period #{date1.strftime("%d/%m/%Y")} to #{date2.strftime("%d/%m/%Y")}"

(date1..date2).each do |date|
  p date.strftime("%d/%m/%Y")
end

# "01/06/2016"
# "02/06/2016"
# "03/06/2016"
# "04/06/2016"
# "05/06/2016"
```

Chapter 35: Modules

Section 35.1: A simple mixin with include

```
module SomeMixin
  def foo
    puts "foo!"
  end
end

class Bar
  include SomeMixin
  def baz
    puts "baz!"
  end
end

b = Bar.new
b.baz          # => "baz!"
b.foo          # => "foo!"
# works thanks to the mixin
```

Now `Bar` is a mix of its own methods and the methods from `SomeMixin`.

Note that how a mixin is used in a class depends on how it is added:

- the `include` keyword evaluates the module code in the class context (eg. method definitions will be methods on instances of the class),
- `extend` will evaluate the module code in the context of the singleton class of the object (methods are available directly on the extended object).

Section 35.2: Modules and Class Composition

You can use Modules to build more complex classes through *composition*. The `include ModuleName` directive incorporates a module's methods into a class.

```
module Foo
  def foo_method
    puts 'foo_method called!'
  end
end

module Bar
  def bar_method
    puts 'bar_method called!'
  end
end

class Baz
  include Foo
  include Bar

  def baz_method
    puts 'baz_method called!'
  end
end
```

`Baz` now contains methods from both `Foo` and `Bar` in addition to its own methods.

```
new_baz = Baz.new
new_baz.baz_method #=> 'baz_method called!'
new_baz.bar_method #=> 'bar_method called!'
new_baz.foo_method #=> 'foo_method called!'
```

Section 35.3: Module as Namespace

Modules can contain other modules and classes:

```
module Namespace

    module Child

        class Foo; end

    end # module Child

    # Foo can now be accessed as:
    #
    Child::Foo

end # module Namespace

# Foo must now be accessed as:
#
Namespace::Child::Foo
```

Section 35.4: A simple mixin with extend

A mixin is just a module that can be added (mixed in) to a class. one way to do it is with the extend method. The `extend` method adds methods of the mixin as class methods.

```
module SomeMixin
  def foo
    puts "foo!"
  end
end

class Bar
  extend SomeMixin
  def baz
    puts "baz!"
  end
end

b = Bar.new
b.baz         # => "baz!"
b.foo         # NoMethodError, as the method was NOT added to the instance
Bar.foo       # => "foo!"
# works only on the class itself
```

Chapter 36: Introspection in Ruby

What is introspection?

Introspection is looking inward to know about the inside. That is a simple definition of introspection.

In programming and Ruby in general…introspection is the ability to look at object, class… at run time to know about that one.

Section 36.1: Introspection of class

Lets following are the class definition

class A def a; end end module B def b; end end class C < A include B def c; end end

What are the instance methods of `C`?

```
C.instance_methods # [:c, :b, :a, :to_json, :instance_of?...]
```

What are the instance methods that declare only on `C`?

```
C.instance_methods(false) # [:c]
```

What are the ancestors of class `C`?

```
C.ancestors # [C, B, A, Object,...]
```

Superclass of `C`?

```
C.superclass # A
```

Section 36.2: Lets see some examples

Example:

```
s = "Hello"  # s is a string
```

Then we find out something about s. Lets begin:

So you want to know what is the class of s at run time?

```
irb(main):055:0* s.class
=> String
```

Ohh, good. But what are the methods of s?

```
irb(main):002:0> s.methods
=> [:unicode_normalize, :include?, :to_c, :unicode_normalize!, :unicode_normalized?, :%, :*, :+,
:count, :partition, :unpack, :encode, :encode!, :next, :casecmp, :insert, :bytesize, :match,
:succ!, :next!, :upto, :index, :rindex, :replace, :clear, :chr, :+@, :-@, :setbyte, :getbyte, :<=>,
:<<, :scrub, :scrub!, :byteslice, :==, :===, :dump, :=~, :downcase, :[], :[]=, :upcase, :downcase!,
:capitalize, :swapcase, :upcase!, :oct, :empty?, :eql?, :hex, :chars, :split, :capitalize!,
:swapcase!, :concat, :codepoints, :reverse, :lines, :bytes, :prepend, :scan, :ord, :reverse!,
:center, :sub, :freeze, :inspect, :intern, :end_with?, :gsub, :chop, :crypt, :gsub!, :start_with?,
:rstrip, :sub!, :ljust, :length, :size, :strip!, :succ, :rstrip!, :chomp, :strip, :rjust, :lstrip!,
:tr!, :chomp!, :squeeze, :lstrip, :tr_s!, :to_str, :to_sym, :chop!, :each_byte, :each_char,
:each_codepoint, :to_s, :to_i, :tr_s, :delete, :encoding, :force_encoding, :sum, :delete!,
```

```
:squeeze!, :tr, :to_f, :valid_encoding?, :slice, :slice!, :rpartition, :each_line, :b,
:ascii_only?, :hash, :to_r, :<, :>, :<=, :>=, :between?, :instance_of?, :public_send,
:instance_variable_get, :instance_variable_set, :instance_variable_defined?,
:remove_instance_variable, :private_methods, :kind_of?, :instance_variables, :tap, :is_a?, :extend,
:to_enum, :enum_for, :!~, :respond_to?, :display, :object_id, :send, :method, :public_method,
:singleton_method, :define_singleton_method, :nil?, :class, :singleton_class, :clone, :dup,
:itself, :taint, :tainted?, :untaint, :untrust, :trust, :untrusted?, :methods, :protected_methods,
:frozen?, :public_methods, :singleton_methods, :!, :!=, :__send__, :equal?, :instance_eval,
:instance_exec, :__id__]
```

You want to know if s is an instance of String?

```
irb(main):017:0*
irb(main):018:0* s.instance_of?(String)
=> true
```

What are the public methods of s?

```
irb(main):026:0* s.public_methods
=> [:unicode_normalize, :include?, :to_c, :unicode_normalize!, :unicode_normalized?, :%, :*, :+,
:count, :partition, :unpack, :encode, :encode!, :next, :casecmp, :insert, :bytesize, :match,
:succ!, :next!, :upto, :index, :rindex, :replace, :clear, :chr, :+@, :-@, :setbyte, :getbyte, :<=>,
:<<, :scrub, :scrub!, :byteslice, :==, :===, :dump, :=~, :downcase, :[], :[]=, :upcase, :downcase!,
:capitalize, :swapcase, :upcase!, :oct, :empty?, :eql?, :hex, :chars, :split, :capitalize!,
:swapcase!, :concat, :codepoints, :reverse, :lines, :bytes, :prepend, :scan, :ord, :reverse!,
:center, :sub, :freeze, :inspect, :intern, :end_with?, :gsub, :chop, :crypt, :gsub!, :start_with?,
:rstrip, :sub!, :ljust, :length, :size, :strip!, :succ, :rstrip!, :chomp, :strip, :rjust, :lstrip!,
:tr!, :chomp!, :squeeze, :lstrip, :tr_s!, :to_str, :to_sym, :chop!, :each_byte, :each_char,
:each_codepoint, :to_s, :to_i, :tr_s, :delete, :encoding, :force_encoding, :sum, :delete!,
:squeeze!, :tr, :to_f, :valid_encoding?, :slice, :slice!, :rpartition, :each_line, :b,
:ascii_only?, :hash, :to_r, :<, :>, :<=, :>=, :between?, :pretty_print, :pretty_print_cycle,
:pretty_print_instance_variables, :pretty_print_inspect, :instance_of?, :public_send,
:instance_variable_get, :instance_variable_set, :instance_variable_defined?,
:remove_instance_variable, :private_methods, :kind_of?, :instance_variables, :tap, :pretty_inspect,
:is_a?, :extend, :to_enum, :enum_for, :!~, :respond_to?, :display, :object_id, :send, :method,
:public_method, :singleton_method, :define_singleton_method, :nil?, :class, :singleton_class,
:clone, :dup, :itself, :taint, :tainted?, :untaint, :untrust, :trust, :untrusted?, :methods,
:protected_methods, :frozen?, :public_methods, :singleton_methods, :!, :!=, :__send__, :equal?,
:instance_eval, :instance_exec, :__id__]
```

and private methods....

```
irb(main):030:0* s.private_methods
=> [:initialize, :initialize_copy, :DelegateClass, :default_src_encoding, :irb_binding, :sprintf,
:format, :Integer, :Float, :String, :Array, :Hash, :catch, :throw, :loop, :block_given?, :Complex,
:set_trace_func, :trace_var, :untrace_var, :at_exit, :Rational, :caller, :caller_locations,
:select, :test, :fork, :exit, :`, :gem_original_require, :sleep, :pp, :respond_to_missing?, :load,
:exec, :exit!, :system, :spawn, :abort, :syscall, :printf, :open, :putc, :print, :readline, :puts,
:p, :srand, :readlines, :gets, :rand, :proc, :lambda, :trap, :initialize_clone, :initialize_dup,
:gem, :require, :require_relative, :autoload, :autoload?, :binding, :local_variables, :warn,
:raise, :fail, :global_variables, :__method__, :__callee__, :__dir__, :eval, :iterator?,
:method_missing, :singleton_method_added, :singleton_method_removed, :singleton_method_undefined]
```

Yes, do s have a method name upper. You want to get the upper case version of s? Lets try:

```
irb(main):044:0> s.respond_to?(:upper)
=> false
```

Look like not, the correct method is upcase lets check:

```
irb(main):047:0*
irb(main):048:0* s.respond_to?(:upcase)
=> true
```

Chapter 37: Monkey Patching in Ruby

Monkey Patching is a way of modifying and extending classes in Ruby. Basically, you can modify already defined classes in Ruby, adding new methods and even modifying previously defined methods.

Section 37.1: Changing an existing ruby method

```
puts "Hello readers".reverse # => "sredaer olleH"

class String
  def reverse
    "Hell riders"
  end
end

puts "Hello readers".reverse # => "Hell riders"
```

Section 37.2: Monkey patching a class

Monkey patching is the modification of classes or objects outside of the class itself.

Sometimes it is useful to add custom functionality.

Example: Override String Class to provide parsing to boolean

```
class String
  def to_b
    self =~ (/^(true|TRUE|True|1)$/i) ? true : false
  end
end
```

As you can see, we add the `to_b()` method to the String class, so we can parse any string to a boolean value.

```
>>'true'.to_b
=> true
>>'foo bar'.to_b
=> false
```

Section 37.3: Monkey patching an object

Like patching of classes, you can also patch single objects. The difference is that only that one instance can use the new method.

Example: Override a string object to provide parsing to boolean

```
s = 'true'
t = 'false'

def s.to_b
  self =~ /true/ ? true : false
end

>> s.to_b
=> true
>> t.to_b
=> undefined method `to_b' for "false":String (NoMethodError)
```

Section 37.4: Safe Monkey patching with Refinements

Since Ruby 2.0, Ruby allows to have safer Monkey Patching with refinements. Basically it allows to limit the Monkey Patched code to only apply when it is requested.

First we create a refinement in a module:

```
module RefiningString
  refine String do
    def reverse
      "Hell riders"
    end
  end
end
```

Then we can decide where to use it:

```
class AClassWithoutMP
  def initialize(str)
    @str = str
  end

  def reverse
    @str.reverse
  end
end

class AClassWithMP
  using RefiningString

  def initialize(str)
    @str = str
  end

  def reverse
    str.reverse
  end
end

AClassWithoutMP.new("hello".reverse # => "olle"
AClassWithMP.new("hello").reverse # "Hell riders"
```

Section 37.5: Changing a method with parameters

You can access the exact same context as the method you override.

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

  def name
    @name
  end
end

puts Boat.new("Doat").name # => "Doat"
```

```ruby
class Boat
  def name
    "☐ #{@name} ☐"
  end
end

puts Boat.new("Moat").name # => "☐ Moat ☐"
```

Section 37.6: Adding Functionality

You can add a method to any class in Ruby, whether it's a builtin or not. The calling object is referenced using **`self`**.

```ruby
class Fixnum
  def plus_one
    self + 1
  end

  def plus(num)
    self + num
  end

  def concat_one
    self.to_s + '1'
  end
end

1.plus_one # => 2
3.plus(5) # => 8
6.concat_one # => '61'
```

Section 37.7: Changing any method

```ruby
def hello
  puts "Hello readers"
end

hello # => "Hello readers"

def hello
  puts "Hell riders"
end

hello # => "Hell riders"
```

Section 37.8: Extending an existing class

```ruby
class String
  def fancy
    "~~~{#{self}}~~~"
  end
end

puts "Dorian".fancy # => "~~~{Dorian}~~~"
```

Chapter 38: Recursion in Ruby

Section 38.1: Tail recursion

Many recursive algorithms can be expressed using iteration. For instance, the greatest common denominator function can be written recursively:

```
def gdc (x, y)
  return x if y == 0
  return gdc(y, x%y)
end
```

or iteratively:

```
def gdc_iter (x, y)
  while y != 0 do
    x, y = y, x%y
  end

  return x
end
```

The two algorithms are equivalent in theory, but the recursive version risks a SystemStackError. However, since the recursive method ends with a call to itself, it could be optimized to avoid a stack overflow. Another way to put it: the recursive algorithm can result in the same machine code as the iterative *if* the compiler knows to look for the recursive method call at the end of the method. Ruby doesn't do tail call optimization by default, but you can turn it on with:

```
RubyVM::InstructionSequence.compile_option = {
  tailcall_optimization: true,
  trace_instruction: false
}
```

In addition to turning on tail-call optimization, you also need to turn off instruction tracing. Unfortunately, these options only apply at compile time, so you either need to `require` the recursive method from another file or `eval` the method definition:

```
RubyVM::InstructionSequence.new(<<-EOF).eval
  def me_myself_and_i
    me_myself_and_i
  end
EOF
me_myself_and_i # Infinite loop, not stack overflow
```

Finally, the final return call must return the method and *only the method*. That means you'll need to re-write the standard factorial function:

```
def fact(x)
  return 1 if x <= 1
  return x*fact(x-1)
end
```

To something like:

```
 def fact(x, acc=1)
```

```
  return acc if x <= 1
  return fact(x-1, x*acc)
end
```

This version passes the accumulated sum via a second (optional) argument that defaults to 1.

Further reading: Tail Call Optimization in Ruby and Tailin' Ruby.

Section 38.2: Recursive function

Let's start with a simple algorithm to see how recursion could be implemented in Ruby.

A bakery has products to sell. Products are in packs. It services orders in packs only. Packaging starts from the largest pack size and then the remaining quantities are filled by next pack sizes available.

For e.g. If an order of 16 is received, bakery allocates 2 from 5 pack and 2 from 3 pack. 2*5+2*3 = 16. Let's see how this is implemented in recursion. "allocate" is the recursive function here.

```
#!/usr/bin/ruby

class Bakery
  attr_accessor :selected_packs

  def initialize
    @packs = [5,3] # pack sizes 5 and 3
    @selected_packs = []
  end

  def allocate(qty)
    remaining_qty = nil

    # ================================================
    # packs are allocated in large packs first order
    # to minimize the packaging space
    # ================================================
    @packs.each do |pack|
      remaining_qty = qty - pack

      if remaining_qty > 0
        ret_val = allocate(remaining_qty)
        if ret_val == 0
          @selected_packs << pack
          remaining_qty = 0
          break
        end
      elsif remaining_qty == 0
        @selected_packs << pack
        break
      end
    end

    remaining_qty
  end
end

bakery = Bakery.new
bakery.allocate(16)
puts "Pack combination is: #{bakery.selected_packs.inspect}"
```

Output is:

> Pack combination is: [3, 3, 5, 5]

Chapter 39: Splat operator (*)

Section 39.1: Variable number of arguments

The splat operator removes individual elements of an array and makes them into a list. This is most commonly used to create a method that accepts a variable number of arguments:

```
# First parameter is the subject and the following parameters are their spouses
def print_spouses(person, *spouses)
  spouses.each do |spouse|
    puts "#{person} married #{spouse}."
  end
end

print_spouses('Elizabeth', 'Conrad', 'Michael', 'Mike', 'Eddie', 'Richard', 'John', 'Larry')
```

Notice that an array only counts as one item on the list, so you will need to us the splat operator on the calling side too if you have an array you want to pass:

```
bonaparte = ['Napoleon','Joséphine','Marie Louise']
print_spouses(*bonaparte)
```

Section 39.2: Coercing arrays into parameter list

Suppose you had an array:

```
pair = ['Jack','Jill']
```

And a method that takes two arguments:

```
def print_pair (a, b)
  puts "#{a} and #{b} are a good couple!"
end
```

You might think you could just pass the array:

```
print_pair(pair) # wrong number of arguments (1 for 2) (ArgumentError)
```

Since the array is just one argument, not two, so Ruby throws an exception. You *could* pull out each element individually:

```
print_pair(pair[0], pair[1])
```

Or you can use the splat operator to save yourself some effort:

```
print_pair(*pair)
```

Chapter 40: JSON with Ruby

Section 40.1: Using JSON with Ruby

JSON (JavaScript Object Notation) is a lightweight data interchange format. Many web applications use it to send and receive data.

In Ruby you can simply work with JSON.

At first you have to `require 'json'`, then you can parse a JSON string via the `JSON.parse()` command.

```
require 'json'

j = '{"a": 1, "b": 2}'
puts JSON.parse(j)
>> {"a"=>1, "b"=>2}
```

What happens here, is that the parser generates a Ruby Hash out of the JSON.

The other way around, generating JSON out of a Ruby hash is as simple as parsing. The method of choice is `to_json`:

```
require 'json'

hash = { 'a' => 1, 'b' => 2 }
json = hash.to_json
puts json
>> {"a":1,"b":2}
```

Section 40.2: Using Symbols

You can use JSON together with Ruby symbols. With the option symbolize_names for the parser, the keys in the resulting hash will be symbols instead of strings.

```
json = '{ "a": 1, "b": 2 }'
puts JSON.parse(json, symbolize_names: true)
>> {:a=>1, :b=>2}
```

Chapter 41: Pure RSpec JSON API testing

Section 41.1: Testing Serializer object and introducing it to Controller

Let say you want to build your API to comply jsonapi.org specification and the result should look like:

```
{
  "article": {
    "id": "305",
    "type": "articles",
    "attributes": {
      "title": "Asking Alexandria"
    }
  }
}
```

Test for Serializer object may look like this:

```
# spec/serializers/article_serializer_spec.rb

require 'rails_helper'

RSpec.describe ArticleSerializer do
  subject { described_class.new(article) }
  let(:article) { instance_double(Article, id: 678, title: "Bring Me The Horizon") }

  describe "#as_json" do
    let(:result) { subject.as_json }

    it 'root should be article Hash' do
      expect(result).to match({
        article: be_kind_of(Hash)
      })
    end

    context 'article hash' do
      let(:article_hash) { result.fetch(:article) }

      it 'should contain type and id' do
        expect(article_hash).to match({
          id: article.id.to_s,
          type: 'articles',
          attributes: be_kind_of(Hash)
        })
      end

      context 'attributes' do
        let(:article_hash_attributes) { article_hash.fetch(:attributes) }

        it do
          expect(article_hash_attributes).to match({
            title: /[Hh]orizon/,
          })
        end
      end
    end
  end
```

```
end
```

Serializer object may look like this:

```
# app/serializers/article_serializer.rb

class ArticleSerializer
  attr_reader :article

  def initialize(article)
    @article = article
  end

  def as_json
    {
      article: {
        id: article.id.to_s,
        type: 'articles',
        attributes: {
          title: article.title
        }
      }
    }
  end
end
```

When we run our "serializers" specs everything passes.

That's pretty boring. Let's introduce a typo to our Article Serializer: Instead of `type: "articles"` let's return `type: "events"` and rerun our tests.

```
rspec spec/serializers/article_serializer_spec.rb

.F.

Failures:

  1) ArticleSerializer#as_json article hash should contain type and id
     Failure/Error:
       expect(article_hash).to match({
         id: article.id.to_s,
         type: 'articles',
         attributes: be_kind_of(Hash)
       })

       expected {:id=>"678", :type=>"event",
:attributes=>{:title=>"Bring Me The Horizon"}} to match {:id=>"678",
:type=>"articles", :attributes=>(be a kind of Hash)}
       Diff:

       @@ -1,4 +1,4 @@
       -:attributes => (be a kind of Hash),
       +:attributes => {:title=>"Bring Me The Horizon"},
        :id => "678",
       -:type => "articles",
       +:type => "events",

     # ./spec/serializers/article_serializer_spec.rb:20:in `block (4
levels) in <top (required)>'
```

Once you've run the test it's pretty easy to spot the error.

Once you fix the error (correct the type to be `article`) you can introduce it to Controller like this:

```
# app/controllers/v2/articles_controller.rb
module V2
  class ArticlesController < ApplicationController
    def show
      render json: serializer.as_json
    end

    private
      def article
        @article ||= Article.find(params[:id])
      end

      def serializer
        @serializer ||= ArticleSerializer.new(article)
      end
  end
end
```

This example is based on article: http://www.eq8.eu/blogs/30-pure-rspec-json-api-testing

Chapter 42: Gem Creation/Management

Section 42.1: Gemspec Files

Each gem has a file in the format of `<gem name>.gemspec` which contains metadata about the gem and it's files. The format of a gemspec is as follows:

```
Gem::Specification.new do |s|
  # Details about gem. They are added in the format:
  s.<detail name> = <detail value>
end
```

The fields required by RubyGems are:

Either `author = string` or `authors = array`

Use `author =` if there is only one author, and `authors =` when there are multiple. For `authors=` use an array which lists the authors names.

```
files = array
```

Here `array` is a list of all the files in the gem. This can also be used with the `Dir[]` function, for example if all your files are in the `/lib/` directory, then you can use `files = Dir["/lib/"]`.

```
name = string
```

Here string is just the name of your gem. Rubygems recommends a few rules you should follow when naming your gem.

1. Use underscores, NO SPACES
2. Use only lowercase letters
3. Use hypens for gem extension (e.g. if your gem is named `example` for an extension you would name it `example-extension`) so that when then extension is required it can be required as `require "example/extension"`.

RubyGems also adds "If you publish a gem on rubygems.org it may be removed if the name is objectionable, violates intellectual property or the contents of the gem meet these criteria. You can report such a gem on the RubyGems Support site."

```
platform=
```

I don't know

```
require_paths=
```

I don't know

```
summary= string
```

String is a summery of the gems purpose and anything that you would like to share about the gem.

```
version= string
```

The current version number of the gem.

The recommended fields are:

```
email = string
```

An email address that will be associated with the gem.

```
homepage= string
```

The website where the gem lives.

Either `license=` or `licenses=`

I don't know

Section 42.2: Building A Gem

Once you have created your gem to publish it you have to follow a few steps:

1. Build your gem with `gem build <gem name>.gemspec` (the gemspec file must exist)
2. Create a RubyGems account if you do not already have one here
3. Check to make sure that no gems exist that share your gems name
4. Publish your gem with `gem publish <gem name>.<gem version number>.gem`

Section 42.3: Dependencies

To list the dependency tree:

```
gem dependency
```

To list which gems depend on a specific gem (bundler for example)

```
gem dependency bundler --reverse-dependencies
```

Chapter 43: rbenv

Section 43.1: Uninstalling a Ruby

There are two ways to uninstall a particular version of Ruby. The easiest is to simply remove the directory from `~/.rbenv/versions`:

```
$ rm -rf ~/.rbenv/versions/2.1.0
```

Alternatively, you can use the uninstall command, which does exactly the same thing:

```
$ rbenv uninstall 2.1.0
```

If this version happens to be in use somewhere, you'll need to update your global or local version. To revert to the version that's first in your path (usually the default provided by your system) use:

```
$ rbenv global system
```

Section 43.2: Install and manage versions of Ruby with rbenv

The easiest way to install and manage various versions of Ruby with rbenv is to use the ruby-build plugin.

First clone the rbenv repository to your home directory:

```
$ git clone https://github.com/rbenv/rbenv.git ~/.rbenv
```

Then clone the ruby-build plugin:

```
$ git clone https://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-build
```

Ensure that rbenv is initialized in your shell session, by adding this to your `.bash_profile` or `.zshrc`:

```
type rbenv > /dev/null
if [ "$?" = "0" ]; then
    eval "$(rbenv init -)"
fi
```

(This essentially first checks if `rbenv` is available, and initializes it).

You will probably have to restart your shell session - or simply open a new Terminal window.

Note: If you're running on OSX, you will also need to install the Mac OS Command Line Tools with:

```
$ xcode-select --install
```

You can also install `rbenv` using Homebrew instead of building from the source:

```
$ brew update
$ brew install rbenv
```

Then follow the instructions given by:

```
$ rbenv init
```

Install a new version of Ruby:

List the versions available with:

```
$ rbenv install --list
```

Choose a version and install it with:

```
$ rbenv install 2.2.0
```

Mark the installed version as the global version - i.e. the one that your system uses by default:

```
$ rbenv global 2.2.0
```

Check what your global version is with:

```
$ rbenv global
=> 2.2.0
```

You can specify a local project version with:

```
$ rbenv local 2.1.2
=> (Creates a .ruby-version file at the current directory with the specified version)
```

Footnotes:

[1]: Understanding PATH

Chapter 44: Gem Usage

Section 44.1: Installing ruby gems

This guide assumes you already have Ruby installed. If you're using Ruby < `1.9` you'll have to manually install RubyGems as it won't be included natively.

To install a ruby gem, enter the command:

```
gem install [gemname]
```

If you are working on a project with a list of gem dependencies, then these will be listed in a file named `Gemfile`. To install a new gem in the project, add the following line of code in the `Gemfile`:

```
gem 'gemname'
```

This `Gemfile` is used by the Bundler gem to install dependencies your project requires, this does however mean that you'll have to install Bundler first by running (if you haven't already):

```
gem install bundler
```

Save the file, and then run the command:

```
bundle install
```

Specifying versions

The version number can be specified on the command live, with the `-v` flag, such as:

```
gem install gemname -v 3.14
```

When specifying version numbers in a `Gemfile`, you have several options available:

- No version specified (`gem 'gemname'`) -- Will install the *latest* version which is compatible with other gems in the `Gemfile`.
- Exact version specified (`gem 'gemname', '3.14'`) -- Will only attempt to install version `3.14` (and fail if this is incompatible with other gems in the `Gemfile`).
- **Optimistic** minimum version number (`gem 'gemname', '>=3.14'`) -- Will only attempt to install the *latest* version which is compatible with other gems in the `Gemfile`, and fails if no version greater than or equal to `3.14` is compatible. The operator `>` can also be used.
- **Pessimistic** minimum version number (`gem 'gemname', '~>3.14'`) -- This is functionally equivalent to using `gem 'gemname', '>=3.14', '<4'`. In other words, only the number after the *final period* is permitted to increase.

As a best practice: You might want to use one of the Ruby version management libraries like rbenv or rvm. Through these libraries, you can install different versions of Ruby runtimes and gems accordingly. So, when working in a project, this will be especially handy because most of the projects are coded against a known Ruby version.

Section 44.2: Gem installation from github/filesystem

You can install a gem from github or filesystem. If the gem has been checked out from git or somehow already on the file system, you could install it using

```
gem install --local path_to_gem/filename.gem
```

Installing gem from github. Download the sources from github

```
mkdir newgem
cd newgem
git clone https://urltogem.git
```

Build the gem

```
gem build GEMNAME.gemspec
gem install gemname-version.gem
```

Section 44.3: Checking if a required gem is installed from within code

To check if a required gem is installed, from within your code, you can use the following (using nokogiri as an example):

```
begin
  found_gem = Gem::Specification.find_by_name('nokogiri')
  require 'nokogiri'
  ....
  <the rest of your code>
rescue Gem::LoadError
end
```

However, this can be further extended to a function that can be used in setting up functionality within your code.

```
def gem_installed?(gem_name)
  found_gem = false
  begin
    found_gem = Gem::Specification.find_by_name(gem_name)
  rescue Gem::LoadError
     return false
  else
    return true
  end
end
```

Now you can check if the required gem is installed, and print an error message.

```
if gem_installed?('nokogiri')
  require 'nokogiri'
else
  printf "nokogiri gem required\n"
  exit 1
end
```

or

```
if gem_installed?('nokogiri')
  require 'nokogiri'
else
  require 'REXML'
end
```

Section 44.4: Using a Gemfile and Bundler

A `Gemfile` is the standard way to organize dependencies in your application. A basic Gemfile will look like this:

```
source 'https://rubygems.org'

gem 'rack'
gem 'sinatra'
gem 'uglifier'
```

You can specify the versions of the gem you want as follows:

```
# Match except on point release. Use only 1.5.X
gem 'rack', '~>1.5.2'
# Use a specific version.
gem 'sinatra', '1.4.7'
# Use at least a version or anything greater.
gem 'uglifier', '>= 1.3.0'
```

You can also pull gems straight from a git repo:

```
# pull a gem from github
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git'
# you can specify a sha
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git', sha:
'30d4fb468fd1d6373f82127d845b153f17b54c51'
# you can also specify a branch, though this is often unsafe
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git', branch: 'master'
```

You can also group gems depending on what they are used for. For example:

```
group :development, :test do
    # This gem is only available in dev and test, not production.
    gem 'byebug'
end
```

You can specify which platform certain gems should run on if you application needs to be able to run on multiple platforms. For example:

```
platform :jruby do
  gem 'activerecord-jdbc-adapter'
  gem 'jdbc-postgres'
end

platform :ruby do
  gem 'pg'
end
```

To install all the gems from a Gemfile do:

```
gem install bundler
bundle install
```

Section 44.5: Bundler/inline (bundler v1.10 and later)

Sometimes you need to make a script for someone but you are not sure what he has on his machine. Is there everything that your script needs? Not to worry. Bundler has a great function called in line.

It provides a `gemfile` method and before the script is run it downloads and requires all the necessary gems. A little example:

```
require 'bundler/inline' #require only what you need

#Start the bundler and in it use the syntax you are already familiar with
gemfile(true) do
  source 'https://rubygems.org'
        gem 'nokogiri', '~> 1.6.8.1'
        gem 'ruby-graphviz'
end
```

Chapter 45: Singleton Class

Section 45.1: Introduction

Ruby has three types of objects:

- Classes and modules which are instances of class Class or class Module.
- Instances of classes.
- Singleton Classes.

Each object has a class which contains its methods:

```
class Example
end

object = Example.new

object.class  # => Example
Example.class # => Class
Class.class   # => Class
```

Objects themselves can't contain methods, only their class can. But with singleton classes, it is possible to add methods to any object including other singleton classes.

```
def object.foo
  :foo
end
object.foo #=> :foo
```

`foo` is defined on singleton class of `object`. Other `Example` instances can not reply to `foo`.

Ruby creates singleton classes on demand. Accessing them or adding methods to them forces Ruby to create them.

Section 45.2: Inheritance of Singleton Class

Subclassing also Subclasses Singleton Class

```
class Example
end

Example.singleton_class #=> #<Class:Example>

def Example.foo
  :example
end

class SubExample < Example
end

SubExample.foo #=> :example

SubExample.singleton_class.superclass #=> #<Class:Example>
```

Extending or Including a Module does not Extend Singleton Class

```
module ExampleModule
end
```

```ruby
def ExampleModule.foo
  :foo
end

class Example
  extend ExampleModule
  include ExampleModule
end

Example.foo #=> NoMethodError: undefined method
```

Section 45.3: Singleton classes

All objects are instances of a class. However, that is not the whole truth. In Ruby, every object also has a somewhat hidden *singleton class*.

This is what allows methods to be defined on individual objects. The singleton class sits between the object itself and its actual class, so all methods defined on it are available for that object, and that object only.

```ruby
object = Object.new

def object.exclusive_method
  'Only this object will respond to this method'
end

object.exclusive_method
# => "Only this object will respond to this method"

Object.new.exclusive_method rescue $!
# => #<NoMethodError: undefined method `exclusive_method' for #<Object:0xa17b77c>>
```

The example above could have been written using `define_singleton_method`:

```ruby
object.define_singleton_method :exclusive_method do
  "The method is actually defined in the object's singleton class"
end
```

Which is the same as defining the method on `object`'s `singleton_class`:

```ruby
# send is used because define_method is private
object.singleton_class.send :define_method, :exclusive_method do
  "Now we're defining an instance method directly on the singleton class"
end
```

Before the existence of `singleton_class` as part of Ruby's core API, singleton classes were known as *metaclasses* and could be accessed via the following idiom:

```ruby
class << object
  self  # refers to object's singleton_class
end
```

Section 45.4: Message Propagation with Singleton Class

Instances never contain a method they only carry data. However we can define a singleton class for any object including an instance of a class.

When a message is passed to an object (method is called) Ruby first checks if a singleton class is defined for that

object and if it can reply to that message otherwise Ruby checks instance's class' ancestors chain and walks up on that.

```
class Example
  def foo
    :example
  end
end

Example.new.foo #=> :example

module PrependedModule
  def foo
    :prepend
  end
end

class Example
  prepend PrependedModule
end

Example.ancestors #=> [Prepended, Example, Object, Kernel, BasicObject]
e = Example.new
e.foo #=> :prepended

def e.foo
  :singleton
end

e.foo #=> :singleton
```

Section 45.5: Reopening (monkey patching) Singleton Classes

There are three ways to reopen a Singleton Class

- Using `class_eval` on a singleton class.
- Using `class <<` block.
- Using `def` to define a method on the object's singleton class directly

```
class Example
end

Example.singleton_class.class_eval do
  def foo
    :foo
  end
end

Example.foo #=> :foo
```

```
class Example
end

class << Example
  def bar
    :bar
  end
end

Example.bar #=> :bar
```

```ruby
class Example
end

def Example.baz
  :baz
end

Example.baz #=> :baz
```

Every object has a singleton class which you can access

```ruby
class Example
end
ex1 = Example.new
def ex1.foobar
  :foobar
end
ex1.foobar #=> :foobar

ex2 = Example.new
ex2.foobar #=> NoMethodError
```

Section 45.6: Accessing Singleton Class

There are two ways to get singleton class of an object

- `singleton_class` method.
- Reopening singleton class of an object and returning `self`.

```ruby
object.singleton_class
```

```ruby
singleton_class = class << object
  self
end
```

Section 45.7: Accessing Instance/Class Variables in Singleton Classes

Singleton classes share their instance/class variables with their object.

```ruby
class Example
  @@foo = :example
end

def Example.foo
  class_variable_get :@@foo
end

Example.foo #=> :example
```

```ruby
class Example
  def initialize
    @foo = 1
  end

  def foo
    @foo
  end
end
```

```ruby
e = Example.new

e.instance_eval <<-BLOCK
  def self.increase_foo
    @foo += 1
  end
BLOCK

e.increase_foo
e.foo #=> 2
```

Blocks close around their instance/class variables target. Accessing instance or class variables using a block in `class_eval` or `instance_eval` isn't possible. Passing a string to `class_eval` or using `class_variable_get` works around the problem.

```ruby
class Foo
  @@foo = :foo
end

class Example
  @@foo = :example

  Foo.define_singleton_method :foo do
    @@foo
  end
end

Foo.foo #=> :example
```

Chapter 46: Queue

Section 46.1: Multiple Workers One Sink

We want to gather data created by multiple Workers.

First we create a Queue:

```
sink = Queue.new
```

Then 16 workers all generating a random number and pushing it into sink:

```
(1..16).to_a.map do
  Thread.new do
    sink << rand(1..100)
  end
end.map(&:join)
```

And to get the data, convert a Queue to an Array:

```
data = [].tap { |a| a << sink.pop until sink.empty? }
```

Section 46.2: Converting a Queue into an Array

```
q = Queue.new
q << 1
q << 2

a = Array.new
a << q.pop until q.empty?
```

Or a one liner:

```
[].tap { |array| array < queue.pop until queue.empty? }
```

Section 46.3: One Source Multiple Workers

We want to process data in parallel.

Let's populate source with some data:

```
source = Queue.new
data = (1..100)
data.each { |e| source << e }
```

Then create some workers to process data:

```
(1..16).to_a.map do
  Thread.new do
    until source.empty?
      item = source.pop
      sleep 0.5
      puts "Processed: #{item}"
    end
  end
```

```
end.map(&:join)
```

Section 46.4: One Source - Pipeline of Work - One Sink

We want to process data in parallel and push it down the line to be processed by other workers.

Since Workers both consume and produce data we have to create two queues:

```
first_input_source = Queue.new
first_output_sink  = Queue.new
100.times { |i| first_input_source << i }
```

First wave of workers read an item from `first_input_source`, process the item, and write results in `first_output_sink`:

```
(1..16).to_a.map do
  Thread.new do
    loop do
      item = first_input_source.pop
      first_output_source << item ** 2
      first_output_source << item ** 3
    end
  end
end
```

Second wave of workers uses `first_output_sink` as its input source and reads, process then writes to another output sink:

```
second_input_source = first_output_sink
second_output_sink  = Queue.new

(1..32).to_a.map do
  Thread.new do
    loop do
      item = second_input_source.pop
      second_output_sink << item * 2
      second_output_sink << item * 3
    end
  end
end
```

Now `second_output_sink` is the sink, let's convert it to an array:

```
sleep 5 # workaround in place of synchronization
sink = second_output_sink
[].tap { |a| a << sink.pop until sink.empty? }
```

Section 46.5: Pushing Data into a Queue - #push

```
q = Queue.new
q << "any object including another queue"
# or
q.push :data
```

- There is no high water mark, queues can infinitely grow.
- *#push* never blocks

Section 46.6: Pulling Data from a Queue - #pop

```
q = Queue.new
q << :data
q.pop #=> :data
```

- `#pop` will block until there is some data available.
- `#pop` can be used for synchronization.

Section 46.7: Synchronization - After a Point in Time

```
syncer = Queue.new

a = Thread.new do
  syncer.pop
  puts "this happens at end"
end

b = Thread.new do
  puts "this happens first"
  STDOUT.flush
  syncer << :ok
end

[a, b].map(&:join)
```

Section 46.8: Merging Two Queues

- To avoid infinitely blocking, reading from queues shouldn't happen on the thread merge is happening on.
- To avoid synchronization or infinitely waiting for one of queues while other has data, reading from queues shouldn't happen on same thread.

Let's start by defining and populating two queues:

```
q1 = Queue.new
q2 = Queue.new
(1..100).each { |e| q1 << e }
(101..200).each { |e| q2 << e }
```

We should create another queue and push data from other threads into it:

```
merged = Queue.new

[q1, q2].map do |q|
  Thread.new do
    loop do
      merged << q.pop
    end
  end
end
```

If you know you can completely consume both queues (consumption speed is higher than production, you won't run out of RAM) there is a simpler approach:

```
merged = Queue.new
merged << q1.pop until q1.empty?
```

```
merged << q2.pop until q2.empty?
```

Chapter 47: Destructuring

Section 47.1: Overview

Most of the magic of destructuring uses the splat (`*`) operator.

Example	Result / *comment*
`a, b = [0,1]`	`a=0, b=1`
`a, *rest = [0,1,2,3]`	`a=0, rest=[1,2,3]`
`a, * = [0,1,2,3]`	`a=0` *Equivalent to* `.first`
`*, z = [0,1,2,3]`	`z=3` *Equivalent to* `.last`

Section 47.2: Destructuring Block Arguments

```
triples = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

triples.each { |(first, second, third)| puts second }
# 2
# 5
# 8

triples.map { |(first, *rest)| rest.join(' ') } # => ["2 3", "5 6", "8 9"]
```

Chapter 48: Struct

Section 48.1: Creating new structures for data

`Struct` defines new classes with the specified attributes and accessor methods.

```
Person = Struct.new :first_name, :last_name
```

You can then instantiate objects and use them:

```
person = Person.new 'John', 'Doe'
# => #<struct Person first_name="John", last_name="Doe">

person.first_name
# => "John"

person.last_name
# => "Doe"
```

Section 48.2: Customizing a structure class

```
Person = Struct.new :name do
  def greet(someone)
    "Hello #{someone}! I am #{name}!"
  end
end

Person.new('Alice').greet 'Bob'
# => "Hello Bob! I am Alice!"
```

Section 48.3: Attribute lookup

Attributes can be accessed strings and symbols as keys. Numerical indexes also work.

```
Person = Struct.new :name
alice = Person.new 'Alice'

alice['name']  # => "Alice"
alice[:name]   # => "Alice"
alice[0]       # => "Alice"
```

Chapter 49: Metaprogramming

Metaprogramming can be described in two ways:

"Computer programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime".

More simply put: **Metaprogramming is writing code that writes code during runtime to make your life easier**.

Section 49.1: Implementing "with" using instance evaluation

Many languages feature a `with` statement that allows programmers to omit the receiver of method calls.

`with` can be easily emulated in Ruby using `instance_eval`:

```
def with(object, &block)
  object.instance_eval &block
end
```

The `with` method can be used to seamlessly execute methods on objects:

```
hash = Hash.new

with hash do
  store :key, :value
  has_key? :key       # => true
  values              # => [:value]
end
```

Section 49.2: send() method

`send()` is used to pass message to `object`. `send()` is an instance method of the `Object` class. The first argument in `send()` is the message that you're sending to the object - that is, the name of a method. It could be `string` or `symbol` but **symbols** are preferred. Then arguments those need to pass in method, those will be the remaining arguments in `send()`.

```
class Hello
  def hello(*args)
    puts 'Hello ' + args.join(' ')
  end
end
h = Hello.new
h.send :hello, 'gentle', 'readers'    #=> "Hello gentle readers"
# h.send(:hello, 'gentle', 'readers') #=> Here :hello is method and rest are the arguments to method.
```

Here is the more descriptive example

```
class Account
  attr_accessor :name, :email, :notes, :address

  def assign_values(values)
    values.each_key do |k, v|
      # How send method would look a like
      # self.name = value[k]
      self.send("#{k}=", values[k])
    end
  end
```

```
end

user_info = {
  name: 'Matt',
  email: 'test@gms.com',
  address: '132 random st.',
  notes: "annoying customer"
}

account = Account.new
If attributes gets increase then we would messup the code
#--------- Bad way --------------
account.name = user_info[:name]
account.address = user_info[:address]
account.email = user_info[:email]
account.notes = user_info[:notes]

# --------- Meta Programing way --------------
account.assign_values(user_info) # With single line we can assign n number of attributes

puts account.inspect
```

Note: `send()` itself is not recommended anymore. Use `__send__()` which has the power to call private methods, or (recommended) `public_send()`

Section 49.3: Defining methods dynamically

With Ruby you can modify the structure of the program in execution time. One way to do it, is by defining methods dynamically using the method `method_missing`.

Let's say that we want to be able to test if a number is greater than other number with the syntax `777.is_greater_than_123?`.

```
# open Numeric class
class Numeric
  # override `method_missing`
  def method_missing(method_name,*args)
    # test if the method_name matches the syntax we want
    if method_name.to_s.match /^is_greater_than_(\d+)\?$/
      # capture the number in the method_name
      the_other_number = $1.to_i
      # return whether the number is greater than the other number or not
      self > the_other_number
    else
      # if the method_name doesn't match what we want, let the previous definition of
`method_missing` handle it
      super
    end
  end
end
```

One important thing to remember when using `method_missing` that one should also override `respond_to?` method:

```
class Numeric
   def respond_to?(method_name, include_all = false)
     method_name.to_s.match(/^is_greater_than_(\d+)\?$/) || super
   end
end
```

Forgetting to do so leads to a inconsistent situation, when you can successfully call `600.is_greater_than_123`, but `600.respond_to(:is_greater_than_123)` returns false.

Section 49.4: Defining methods on instances

In ruby you can add methods to existing instances of any class. This allows you to add behavior to and instance of a class without changing the behavior of the rest of the instances of that class.

```
class Example
  def method1(foo)
    puts foo
  end
end

#defines method2 on object exp
exp = Example.new
exp.define_method(:method2) {puts "Method2"}

#with method parameters
exp.define_method(:method3) {|name| puts name}
```

Chapter 50: Dynamic Evaluation

Parameter	Details
`"source"`	Any Ruby source code
`binding`	An instance of `Binding` class
`proc`	An instance of `Proc` class

Section 50.1: Instance evaluation

The `instance_eval` method is available on all objects. It evaluates code in the context of the receiver:

```
object = Object.new

object.instance_eval do
  @variable = :value
end

object.instance_variable_get :@variable  # => :value
```

`instance_eval` sets `self` to `object` for the duration of the code block:

```
object.instance_eval { self == object }  # => true
```

The receiver is also passed to the block as its only argument:

```
object.instance_eval { |argument| argument == object }  # => true
```

The `instance_exec` method differs in this regard: it passes its arguments to the block instead.

```
object.instance_exec :@variable do |name|
  instance_variable_get name  # => :value
end
```

Section 50.2: Evaluating a String

Any `String` can be evaluated at runtime.

```
class Example
  def self.foo
    :foo
  end
end

eval "Example.foo" #=> :foo
```

Section 50.3: Evaluating Inside a Binding

Ruby keeps track of local variables and `self` variable via an object called binding. We can get binding of a scope with calling `Kernel#binding` and evaluate string inside a binding via `Binding#eval`.

```
b = proc do
  local_variable = :local
  binding
end.call
```

```ruby
b.eval "local_variable" #=> :local
```

```ruby
def fake_class_eval klass, source = nil, &block
  class_binding = klass.send :eval, "binding"

  if block
    class_binding.local_variable_set :_fake_class_eval_block, block
    class_binding.eval "_fake_class_eval_block.call"
  else
    class_binding.eval source
  end
end

class Example
end

fake_class_eval Example, <<-BLOCK
  def self.foo
    :foo
  end
BLOCK

fake_class_eval Example do
  def bar
    :bar
  end
end

Example.foo #=> :foo
Example.new.bar #=> :bar
```

Section 50.4: Dynamically Creating Methods from Strings

Ruby offers `define_method` as a private method on modules and classes for defining new instance methods. However, the 'body' of the method must be a **`Proc`** or another existing method.

One way to create a method from raw string data is to use **`eval`** to create a Proc from the code:

```ruby
xml = <<ENDXML
<methods>
  <method name="go">puts "I'm going!"</method>
  <method name="stop">7*6</method>
</methods>
ENDXML

class Foo
  def self.add_method(name,code)
    body = eval( "Proc.new{ #{code} }" )
    define_method(name,body)
  end
end

require 'nokogiri' # gem install nokogiri
doc = Nokogiri.XML(xml)
doc.xpath('//method').each do |meth|
  Foo.add_method( meth['name'], meth.text )
end

f = Foo.new
p Foo.instance_methods(false)  #=> [:go, :stop]
p f.public_methods(false)      #=> [:go, :stop]
```

```
f.go                            #=> "I'm going!"
p f.stop                        #=> 42
```

Chapter 51: instance_eval

Parameter	Details
`string`	Contains the Ruby source code to be evaluated.
`filename`	File name to use for error reporting.
`lineno`	Line number to use for error reporting.
`block`	The block of code to be evaluated.
`obj`	The receiver is passed to the block as its only argument.

Section 51.1: Instance evaluation

The `instance_eval` method is available on all objects. It evaluates code in the context of the receiver:

```
object = Object.new

object.instance_eval do
  @variable = :value
end

object.instance_variable_get :@variable # => :value
```

`instance_eval` sets `self` to `object` for the duration of the code block:

```
object.instance_eval { self == object } # => true
```

The receiver is also passed to the block as its only argument:

```
object.instance_eval { |argument| argument == object } # => true
```

The `instance_exec` method differs in this regard: it passes its arguments to the block instead.

```
object.instance_exec :@variable do |name|
  instance_variable_get name # => :value
end
```

Section 51.2: Implementing with

Many languages feature a `with` statement that allows programmers to omit the receiver of method calls.

`with` can be easily emulated in Ruby using `instance_eval`:

```
def with(object, &block)
  object.instance_eval &block
end
```

The `with` method can be used to seamlessly execute methods on objects:

```
hash = Hash.new

with hash do
  store :key, :value
  has_key? :key       # => true
  values              # => [:value]
end
```

Chapter 52: Message Passing

Section 52.1: Introduction

In *Object Oriented Design*, objects *receive* messages and *reply* to them. In Ruby, sending a message is *calling a method* and result of that method is the reply.

In Ruby message passing is dynamic. When a message arrives rather than knowing exactly how to reply to it Ruby uses a predefined set of rules to find a method that can reply to it. We can use these rules to interrupt and reply to the message, send it to another object or modify it among other actions.

Each time an object receives a message Ruby checks:

1. If this object has a singleton class and it can reply to this message.
2. Looks up this object's class then class' ancestors chain.
3. One by one checks if a method is available on this ancestor and moves up the chain.

Section 52.2: Message Passing Through Inheritance Chain

```
class Example
  def example_method
    :example
  end

  def subexample_method
    :example
  end

  def not_missed_method
    :example
  end

  def method_missing name
    return :example if name == :missing_example_method
    return :example if name == :missing_subexample_method
    return :subexample if name == :not_missed_method
    super
  end
end

class SubExample < Example
  def subexample_method
    :subexample
  end

  def method_missing name
    return :subexample if name == :missing_subexample_method
    return :subexample if name == :not_missed_method
    super
  end
end

s = Subexample.new
```

To find a suitable method for `SubExample#subexample_method` Ruby first looks at ancestors chain of `SubExample`

```
SubExample.ancestors # => [SubExample, Example, Object, Kernel, BasicObject]
```

It starts from `SubExample`. If we send `subexample_method` message Ruby chooses the one available one SubExample and ignores `Example#subexample_method`.

```
s.subexample_method # => :subexample
```

After `SubExample` it checks `Example`. If we send `example_method` Ruby checks if `SubExample` can reply to it or not and since it can't Ruby goes up the chain and looks into `Example`.

```
s.example_method # => :example
```

After Ruby checks all defined methods then it runs `method_missing` to see if it can reply or not. If we send `missing_subexample_method` Ruby won't be able to find a defined method on `SubExample` so it moves up to `Example`. It can't find a defined method on `Example` or any other class higher in chain either. Ruby starts over and runs `method_missing`. `method_missing` of `SubExample` can reply to `missing_subexample_method`.

```
s.missing_subexample_method # => :subexample
```

However if a method is defined Ruby uses defined version even if it is higher in the chain. For example if we send `not_missed_method` even though `method_missing` of `SubExample` can reply to it Ruby walks up on `SubExample` because it doesn't have a defined method with that name and looks into `Example` which has one.

```
s.not_missed_method # => :example
```

Section 52.3: Message Passing Through Module Composition

Ruby moves up on ancestors chain of an object. This chain can contain both modules and classes. Same rules about moving up the chain apply to modules as well.

```
class Example
end

module Prepended
  def initialize *args
    return super :default if args.empty?
    super
  end
end

module FirstIncluded
  def foo
    :first
  end
end

module SecondIncluded
  def foo
    :second
  end
end

class SubExample < Example
  prepend Prepended
  include FirstIncluded
  include SecondIncluded

  def initialize data = :subexample
    puts data
```

```
  end
end

SubExample.ancestors # => [Prepended, SubExample, SecondIncluded, FirstIncluded, Example, Object,
Kernel, BasicObject]

s = SubExample.new # => :default
s.foo # => :second
```

Section 52.4: Interrupting Messages

There are two ways to interrupt messages.

- Use `method_missing` to interrupt any non defined message.
- Define a method in middle of a chain to intercept the message

After interrupting messages, it is possible to:

- Reply to them.
- Send them somewhere else.
- Modify the message or its result.

Interrupting via `method_missing` and replying to message:

```
class Example
  def foo
    @foo
  end

  def method_missing name, data
    return super unless name.to_s =~ /=$/
    name = name.to_s.sub(/=$/, "")
    instance_variable_set "@#{name}", data
  end
end

e = Example.new

e.foo = :foo
e.foo # => :foo
```

Intercepting message and modifying it:

```
class Example
  def initialize title, body
  end
end

class SubExample < Example
end
```

Now let's imagine our data is "title:body" and we have to split them before calling `Example`. We can define `initialize` on `SubExample`.

```
class SubExample < Example
  def initialize raw_data
    processed_data = raw_data.split ":"
```

```
    super processed_data[0], processed_data[1]
  end
end
```

Intercepting message and sending it to another object:

```
class ObscureLogicProcessor
  def process data
    :ok
  end
end

class NormalLogicProcessor
  def process data
    :not_ok
  end
end

class WrapperProcessor < NormalLogicProcessor
  def process data
    return ObscureLogicProcessor.new.process data if data.obscure?

    super
  end
end
```

Chapter 53: Keyword Arguments

Section 53.1: Using arbitrary keyword arguments with splat operator

You can define a method to accept an arbitrary number of keyword arguments using the *double splat* (`**`) operator:

```
def say(**args)
  puts args
end

say foo: "1", bar: "2"
# {:foo=>"1", :bar=>"2"}
```

The arguments are captured in a `Hash`. You can manipulate the `Hash`, for example to extract the desired arguments.

```
def say(**args)
  puts args[:message] || "Message not found"
end

say foo: "1", bar: "2", message: "Hello World"
# Hello World

say foo: "1", bar: "2"
# Message not found
```

Using a the splat operator with keyword arguments will prevent keyword argument validation, the method will never raise an `ArgumentError` in case of unknown keyword.

As for the standard splat operator, you can re-convert a `Hash` into keyword arguments for a method:

```
def say(message: nil, before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

args = { message: "Hello World", after: "</p><hr>" }
say(**args)
# <p>Hello World</p><hr>

args = { message: "Hello World", foo: "1" }
say(**args)
# => ArgumentError: unknown keyword: foo
```

This is generally used when you need to manipulate incoming arguments, and pass them to an underlying method:

```
def inner(foo:, bar:)
  puts foo, bar
end

def outer(something, foo: nil, bar: nil, baz: nil)
  puts something
  params = {}
  params[:foo] = foo || "Default foo"
  params[:bar] = bar || "Default bar"
  inner(**params)
end
```

```ruby
outer "Hello:", foo: "Custom foo"
# Hello:
# Custom foo
# Default bar
```

Section 53.2: Using keyword arguments

You define a keyword argument in a method by specifying the name in the method definition:

```ruby
def say(message: "Hello World")
  puts message
end

say
# => "Hello World"

say message: "Today is Monday"
# => "Today is Monday"
```

You can define multiple keyword arguments, the definition order is irrelevant:

```ruby
def say(message: "Hello World", before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

say
# => "<p>Hello World</p>"

say message: "Today is Monday"
# => "<p>Today is Monday</p>"

say after: "</p><hr>", message: "Today is Monday"
# => "<p>Today is Monday</p><hr>"
```

Keyword arguments can be mixed with positional arguments:

```ruby
def say(message, before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

say "Hello World", before: "<span>", after: "</span>"
# => "<span>Hello World</span>"
```

Mixing keyword argument with positional argument was a very common approach before Ruby 2.1, because it was not possible to define required keyword arguments.

Moreover, in Ruby < 2.0, it was very common to add an `Hash` at the end of a method definition to use for optional arguments. The syntax is very similar to keyword arguments, to the point where optional arguments via `Hash` are compatible with Ruby 2 keyword arguments.

```ruby
def say(message, options = {})
  before = option.fetch(:before, "<p>")
  after  = option.fetch(:after, "</p>")
  puts "#{before}#{message}#{after}"
end

# The method call is syntactically equivalent to the keyword argument one
say "Hello World", before: "<span>", after: "</span>"
```

```
# => "<span>Hello World</span>"
```

Note that trying to pass a not-defined keyword argument will result in an error:

```
def say(message: "Hello World")
  puts message
end

say foo: "Hello"
# => ArgumentError: unknown keyword: foo
```

Section 53.3: Required keyword arguments

Version ≥ 2.1

Required keyword arguments were introduced in Ruby 2.1, as an improvement to keyword arguments.

To define a keyword argument as required, simply declare the argument without a default value.

```
def say(message:)
  puts message
end

say
# => ArgumentError: missing keyword: message

say message: "Hello World"
# => "Hello World"
```

You can also mix required and non-required keyword arguments:

```
def say(before: "<p>", message:, after: "</p>")
  puts "#{before}#{message}#{after}"
end

say
# => ArgumentError: missing keyword: message

say message: "Hello World"
# => "<p>Hello World</p>"

say message: "Hello World", before: "<span>", after: "</span>"
# => "<span>Hello World</span>"
```

Chapter 54: Truthiness

Section 54.1: All objects may be converted to booleans in Ruby

Use the double negation syntax to check for truthiness of values. All values correspond to a boolean, irrespective of their type.

```
irb(main):001:0> !!1234
=> true
irb(main):002:0> !!"Hello, world!"
(irb):2: warning: string literal in condition
=> true
irb(main):003:0> !!true
=> true
irb(main):005:0> !!{a:'b'}
=> true
```

All values except `nil` and `false` are truthy.

```
irb(main):006:0> !!nil
=> false
irb(main):007:0> !!false
=> false
```

Section 54.2: Truthiness of a value can be used in if-else constructs

You do not need to use double negation in if-else statements.

```
if 'hello'
    puts 'hey!'
else
    puts 'bye!'
end
```

The above code prints 'hey!' on the screen.

Chapter 55: Implicit Receivers and Understanding Self

Section 55.1: There is always an implicit receiver

In Ruby, there is always an implicit receiver for all method calls. The language keeps a reference to the current implicit receiver stored in the variable `self`. Certain language keywords like `class` and `module` will change what `self` points to. Understanding these behaviors is very helpful in mastering the language.

For example, when you first open `irb`

```
irb(main):001:0> self
=> main
```

In this case the `main` object is the implicit receiver (see http://stackoverflow.com/a/917842/417872 for more about `main`).

You can define methods on the implicit receiver using the `def` keyword. For example:

```
irb(main):001:0> def foo(arg)
irb(main):002:1> arg.to_s
irb(main):003:1> end
=> :foo
irb(main):004:0> foo 1
=> "1"
```

This has defined the method foo on the instance of main object running in your repl.

Note that local variables are looked up before method names, so that if you define a local variable with the same name, its reference will supersede the method reference. Continuing from the previous example:

```
irb(main):005:0> defined? foo
=> "method"
irb(main):006:0> foo = 1
=> 1
irb(main):007:0> defined? foo
=> "local-variable"
irb(main):008:0> foo
=> 1
irb(main):009:0> method :foo
=> #<Method: Object#foo>
```

The `method` method can still find the `foo` method because it doesn't check for local variables, while the normal reference `foo` does.

Section 55.2: Keywords change the implicit receiver

When you define a class or module, the implicit receiver becomes a reference to the class itself. For example:

```
puts "I am #{self}"
class Example
  puts "I am #{self}"
end
```

Executing the above code will print:

```
"I am main"
"I am Example"
```

Section 55.3: When to use self?

Most Ruby code utilizes the implicit receiver, so programmers who are new to Ruby are often confused about when to use `self`. The practical answer is that `self` is used in two major ways:

1. To change the receiver.

Ordinarily the behavior of `def` inside a class or module is to create instance methods. Self can be used to define methods on the class instead.

```
class Foo
  def bar
    1
  end

  def self.bar
    2
  end
end

Foo.new.bar #=> 1
Foo.bar #=> 2
```

2. To disambiguate the receiver

When local variables may have the same name as a method an explicit receiver may be required to disambiguate.

Examples:

```
class Example
  def foo
    1
  end

  def bar
    foo + 1
  end

  def baz(foo)
    self.foo + foo # self.foo is the method, foo is the local variable
  end

  def qux
    bar = 2
    self.bar + bar # self.bar is the method, bar is the local variable
  end
end

Example.new.foo    #=> 1
Example.new.bar    #=> 2
Example.new.baz(2) #=> 3
Example.new.qux    #=> 4
```

The other common case requiring disambiguation involves methods that end in the equals sign. For instance:

```
class Example
  def foo=(input)
    @foo = input
  end

  def get_foo
    @foo
  end

  def bar(input)
    foo = input # will create a local variable
  end

  def baz(input)
    self.foo = input # will call the method
  end
end

e = Example.new
e.get_foo #=> nil
e.foo = 1
e.get_foo #=> 1
e.bar(2)
e.get_foo #=> 1
e.baz(2)
e.get_foo #=> 2
```

Chapter 56: Introspection

Section 56.1: View an object's methods

Inspecting an Object

You can find the public methods an object can respond to using either the `methods` or `public_methods` methods, which return an array of symbols:

```
class Foo
  def bar; 42; end
end
f = Foo.new
def f.yay; 17; end
p f.methods.sort
#=> [:!, :!=, :!~, :<=>, :==, :===, :=~, :__id__, :__send__, :bar, :class, :clone,
#=>  :define_singleton_method, :display, :dup, :enum_for, :eql?, :equal?, :extend,
#=>  :freeze, :frozen?, :hash, :inspect, :instance_eval, :instance_exec,
#=>  :instance_of?, :instance_variable_defined?, :instance_variable_get,
#=>  :instance_variable_set, :instance_variables, :is_a?, :itself, :kind_of?,
#=>  :method, :methods, :nil?, :object_id, :private_methods, :protected_methods,
#=>  :public_method, :public_methods, :public_send, :remove_instance_variable,
#=>  :respond_to?, :send, :singleton_class, :singleton_method, :singleton_methods,
#=>  :taint, :tainted?, :tap, :to_enum, :to_s, :trust, :untaint, :untrust,
#=>  :untrusted?, :yay]
```

For a more targeted list, you can remove methods common to all objects, e.g.

```
p (f.methods - Object.methods).sort
#=> [:bar, :yay]
```

Alternatively, you can pass `false` to `methods` or `public_methods`:

```
p f.methods(false) # public and protected singleton methods of `f`
#=> [:yay]

p f.public_methods(false)
#=> [:yay, :bar]
```

You can find the private and protected methods of an object using `private_methods` and `protected_methods`:

```
p f.private_methods.sort
#=> [:Array, :Complex, :DelegateClass, :Float, :Hash, :Integer, :Rational, :String,
#=>  :__callee__, :__dir__, :__method__, :`, :abort, :at_exit, :autoload, :autoload?,
#=>  :binding, :block_given?, :caller, :caller_locations, :catch,
#=>  :default_src_encoding, :eval, :exec, :exit, :exit!, :fail, :fork, :format, :gem,
#=>  :gem_original_require, :gets, :global_variables, :initialize, :initialize_clone,
#=>  :initialize_copy, :initialize_dup, :irb_binding, :iterator?, :lambda, :load,
#=>  :local_variables, :loop, :method_missing, :open, :p, :print, :printf, :proc,
#=>  :putc, :puts, :raise, :rand, :readline, :readlines, :require, :require_relative,
#=>  :respond_to_missing?, :select, :set_trace_func, :singleton_method_added,
#=>  :singleton_method_removed, :singleton_method_undefined, :sleep, :spawn,
#=>  :sprintf, :srand, :syscall, :system, :test, :throw, :trace_var, :trap,
#=>  :untrace_var, :warn]

p f.protected_methods
#=> []
```

As with `methods` and `public_methods`, you can pass `false` to `private_methods` and `protected_methods` to trim away inherited methods.

Inspecting a Class or Module

In addition to `methods`, `public_methods`, `protected_methods`, and `private_methods`, classes and modules expose `instance_methods`, `public_instance_methods`, `protected_instance_methods`, and `private_instance_methods` to determine the methods exposed for objects that inherit from the class or module. As above, you can pass `false` to these methods to exclude inherited methods:

```
p Foo.instance_methods.sort
#=> [:!, :!=, :!~, :<=>, :==, :===, :=~, :__id__, :__send__, :bar, :class,
#=>  :clone, :define_singleton_method, :display, :dup, :enum_for, :eql?,
#=>  :equal?, :extend, :freeze, :frozen?, :hash, :inspect, :instance_eval,
#=>  :instance_exec, :instance_of?, :instance_variable_defined?,
#=>  :instance_variable_get, :instance_variable_set, :instance_variables,
#=>  :is_a?, :itself, :kind_of?, :method, :methods, :nil?, :object_id,
#=>  :private_methods, :protected_methods, :public_method, :public_methods,
#=>  :public_send, :remove_instance_variable, :respond_to?, :send,
#=>  :singleton_class, :singleton_method, :singleton_methods, :taint,
#=>  :tainted?, :tap, :to_enum, :to_s, :trust, :untaint, :untrust, :untrusted?]

p Foo.instance_methods(false)
#=> [:bar]
```

Finally, if you forget the names of most of these in the future, you can find all of these methods using `methods`:

```
p f.methods.grep(/methods/)
#=> [:private_methods, :methods, :protected_methods, :public_methods,
#=>  :singleton_methods]

p Foo.methods.grep(/methods/)
#=> [:public_instance_methods, :instance_methods, :private_instance_methods,
#=>  :protected_instance_methods, :private_methods, :methods,
#=>  :protected_methods, :public_methods, :singleton_methods]
```

Section 56.2: View an object's Instance Variables

It is possible to query an object about its instance variables using `instance_variables`, `instance_variable_defined?`, and `instance_variable_get`, and modify them using `instance_variable_set` and `remove_instance_variable`:

```
class Foo
  attr_reader :bar
  def initialize
    @bar = 42
  end
end
f = Foo.new
f.instance_variables                #=> [:@bar]
f.instance_variable_defined?(:@baz) #=> false
f.instance_variable_defined?(:@bar) #=> true
f.instance_variable_get(:@bar)      #=> 42
f.instance_variable_set(:@bar, 17)   #=> 17
f.bar                               #=> 17
f.remove_instance_variable(:@bar)   #=> 17
f.bar                               #=> nil
f.instance_variables                #=> []
```

The names of instance variables include the `@` symbol. You will get an error if you omit it:

```
f.instance_variable_defined?(:jim)
#=> NameError: `jim' is not allowed as an instance variable name
```

Section 56.3: View Global and Local Variables

The `Kernel` exposes methods for getting the list of `global_variables` and `local_variables`:

```
cats  = 42
$demo = "in progress"
p global_variables.sort
#=> [:$!, :$", :$$, :$&, :$', :$*, :$+, :$,, :$-0, :$-F, :$-I, :$-K, :$-W, :$-a,
#=>  :$-d, :$-i, :$-l, :$-p, :$-v, :$-w, :$., :$/, :$0, :$1, :$2, :$3, :$4, :$5,
#=>  :$6, :$7, :$8, :$9, :$:, :$;, :$<, :$=, :$>, :$?, :$@, :$DEBUG, :$FILENAME,
#=>  :$KCODE, :$LOADED_FEATURES, :$LOAD_PATH, :$PROGRAM_NAME, :$SAFE, :$VERBOSE,
#=>  :$\, :$_, :$`, :$binding, :$demo, :$stderr, :$stdin, :$stdout, :$~]

p local_variables
#=> [:cats]
```

Unlike instance variables there are no methods specifically for getting, setting, or removing global or local variables. Looking for such functionality is usually a sign that your code should be rewritten to use a Hash to store the values. However, if you must modify global or local variables by name, you can use `eval` with a string:

```
var = "$demo"
eval(var)           #=> "in progress"
eval("#{var} = 17")
p $demo             #=> 17
```

By default, `eval` will evaluate your variables in the current scope. To evaluate local variables in a different scope, you must capture the *binding* where the local variables exist.

```
def local_variable_get(name, bound=nil)
  foo = :inside
  eval(name,bound)
end

def test_1
  foo = :outside
  p local_variable_get("foo")
end

def test_2
  foo = :outside
  p local_variable_get("foo",binding)
end

test_1 #=> :inside
test_2 #=> :outside
```

In the above, `test_1` did not pass a binding to `local_variable_get`, and so the `eval` was executed within the context of that method, where a local variable named `foo` was set to `:inside`.

Section 56.4: View Class Variables

Classes and modules have the same methods for introspecting instance variables as any other object. Class and

modules also have similar methods for querying the class variables (`@@these_things`):

```
p Module.methods.grep(/class_variable/)
#=> [:class_variables, :class_variable_get, :remove_class_variable,
#=>  :class_variable_defined?, :class_variable_set]

class Foo
  @@instances = 0
  def initialize
    @@instances += 1
  end
end

class Bar < Foo; end

5.times{ Foo.new }
3.times{ Bar.new }
p Foo.class_variables                   #=> [:@@instances]
p Bar.class_variables                   #=> [:@@instances]
p Foo.class_variable_get(:@@instances)  #=> 8
p Bar.class_variable_get(:@@instances)  #=> 8
```

Similar to instance variables, the name of class variables must begin with `@@`, or you will get an error:

```
p Bar.class_variable_defined?( :instances )
#=> NameError: `instances' is not allowed as a class variable name
```

Chapter 57: Refinements

Section 57.1: Monkey patching with limited scope

Monkey patching's main issue is that it pollutes the global scope. Your code working is at the mercy of all the modules you use not stepping on each others toes. The Ruby solution to this is refinements, which are basically monkey patches in a limited scope.

```
module Patches
  refine Fixnum do
    def plus_one
      self + 1
    end

    def plus(num)
      self + num
    end

    def concat_one
      self.to_s + '1'
    end
  end
end

class RefinementTest
  # has access to our patches
  using Patches

  def initialize
    puts 1.plus_one
    puts 3.concat_one
  end
end

# Main scope doesn't have changes

1.plus_one
# => undefined method `plus_one' for 1:Fixnum (NoMethodError)

RefinementTest.new
# => 2
# => '31'
```

Section 57.2: Dual-purpose modules (refinements or global patches)

It's a good practice to scope patches using Refinements, but sometimes it's nice to load it globally (for example in development, or testing).

Say for example you want to start a console, require your library, and then have the patched methods available in the global scope. You couldn't do this with refinements because `using` needs to be called in a class/module definition. But it's possible to write the code in such a way that it's dual purpose:

```
module Patch
  def patched?; true; end
  refine String do
    include Patch
```

```ruby
  end
end

# globally
String.include Patch
"".patched? # => true

# refinement
class LoadPatch
  using Patch
  "".patched? # => true
end
```

Section 57.3: Dynamic refinements

Refinements have special limitations.

`refine` can only be used in a module scope, but can be programmed using `send :refine`.

`using` is more limited. It can only be called in a class/module definition. Still, it can accept a variable pointing to a module, and can be invoked in a loop.

An example showing these concepts:

```ruby
module Patch
  def patched?; true; end
end

Patch.send(:refine, String) { include Patch }

patch_classes = [Patch]

class Patched
  patch_classes.each { |klass| using klass }
  "".patched? # => true
end
```

Since `using` is so static, there can be issued with load order if the refinement files are not loaded first. A way to address this is to wrap the patched class/module definition in a proc. For example:

```ruby
module Patch
  refine String do
    def patched; true; end
  end
end

class Foo
end

# This is a proc since methods can't contain class definitions
create_patched_class = Proc.new do
  Foo.class_exec do
    class Bar
      using Patch
      def self.patched?; ''.patched == true; end
    end
  end
end
create_patched_class.call
```

```ruby
Foo::Bar.patched? # => true
```

Calling the proc creates the patched class `Foo::Bar`. This can be delayed until after all the code has loaded.

Chapter 58: Catching Exceptions with Begin / Rescue

Section 58.1: A Basic Error Handling Block

Let's make a function to divide two numbers, that's very trusting about its input:

```
def divide(x, y)
  return x/y
end
```

This will work fine for a lot of inputs:

```
> puts divide(10, 2)
5
```

But not all

```
> puts divide(10, 0)
ZeroDivisionError: divided by 0

> puts divide(10, 'a')
TypeError: String can't be coerced into Fixnum
```

We can rewrite the function by wrapping the risky division operation in a `begin... end` block to check for errors, and use a `rescue` clause to output a message and return `nil` if there is a problem.

```
def divide(x, y)
  begin
    return x/y
  rescue
    puts "There was an error"
    return nil
  end
end

> puts divide(10, 0)
There was an error

> puts divide(10, 'a')
There was an error
```

Section 58.2: Saving the Error

You can save the error if you want to use it in the `rescue` clause

```
def divide(x, y)
  begin
    x/y
  rescue => e
    puts "There was a %s (%s)" % [e.class, e.message]
    puts e.backtrace
  end
end

> divide(10, 0)
```

```
There was a ZeroDivisionError (divided by 0)
       from (irb):10:in `/'
       from (irb):10
       from /Users/username/.rbenv/versions/2.3.1/bin/irb:11:in `<main>'

> divide(10, 'a')
There was a TypeError (String can't be coerced into Fixnum)
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/workspace.rb:87:in `eval'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/workspace.rb:87:in `evaluate'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/context.rb:380:in `evaluate'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:489:in `block (2 levels) in eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:623:in `signal_status'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:486:in `block in eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:246:in `block (2 levels) in
each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:232:in `loop'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:232:in `block in
each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:231:in `catch'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:231:in
`each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:485:in `eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:395:in `block in start'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:394:in `catch'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:394:in `start'
/Users/username/.rbenv/versions/2.3.1/bin/irb:11:in `<main>'
```

Section 58.3: Checking for Different Errors

If you want to do different things based on the kind of error, use multiple `rescue` clauses, each with a different error type as an argument.

```ruby
def divide(x, y)
  begin
    return x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
    return nil
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  end
end

> divide(10, 0)
Don't divide by zero!

> divide(10, 'a')
Division only works on numbers!
```

If you want to save the error for use in the `rescue` block:

```ruby
rescue ZeroDivisionError => e
```

Use a `rescue` clause with no argument to catch errors of a type not specified in another `rescue` clause.

```ruby
def divide(x, y)
  begin
    return x/y
```

```
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
    return nil
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  end
end

> divide(nil, 2)
Don't do that (NoMethodError)
```

In this case, trying to divide `nil` by 2 is not a `ZeroDivisionError` or a `TypeError`, so it handled by the default `rescue` clause, which prints out a message to let us know that it was a `NoMethodError`.

Section 58.4: Retrying

In a `rescue` clause, you can use `retry` to run the `begin` clause again, presumably after changing the circumstance that caused the error.

```
def divide(x, y)
  begin
    puts "About to divide..."
    return x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
    y = 1
    retry
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  end
end
```

If we pass parameters that we know will cause a `TypeError`, the `begin` clause is executed (flagged here by printing out "About to divide") and the error is caught as before, and `nil` is returned:

```
> divide(10, 'a')
About to divide...
Division only works on numbers!
 => nil
```

But if we pass parameters that will cause a `ZeroDivisionError`, the `begin` clause is executed, the error is caught, the divisor changed from 0 to 1, and then `retry` causes the `begin` block to be run again (from the top), now with a different y. The second time around there is no error and the function returns a value.

```
> divide(10, 0)
About to divide...     # First time, 10 ÷ 0
Don't divide by zero!
About to divide...     # Second time 10 ÷ 1
=> 10
```

Section 58.5: Checking Whether No Error Was Raised

You can use an `else` clause for code that will be run if no error is raised.

```
def divide(x, y)
  begin
    z = x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  else
    puts "This code will run if there is no error."
    return z
  end
end
```

The `else` clause does not run if there is an error that transfers control to one of the `rescue` clauses:

```
> divide(10,0)
Don't divide by zero!
=> nil
```

But if no error is raised, the `else` clause executes:

```
> divide(10,2)
This code will run if there is no error.
=> 5
```

Note that the `else` clause will not be executed *if you return from the `begin` clause*

```
def divide(x, y)
  begin
    z = x/y
    return z                # Will keep the else clause from running!
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
  else
    puts "This code will run if there is no error."
    return z
  end
end

> divide(10,2)
=> 5
```

Section 58.6: Code That Should Always Run

Use an `ensure` clause if there is code you always want to execute.

```
def divide(x, y)
  begin
    z = x/y
    return z
```

```
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  ensure
    puts "This code ALWAYS runs."
  end
end
```

The `ensure` clause will be executed when there is an error:

```
> divide(10, 0)
Don't divide by zero!   # rescue clause
This code ALWAYS runs.   # ensure clause
=> nil
```

And when there is no error:

```
> divide(10, 2)
This code ALWAYS runs.   # ensure clause
=> 5
```

The ensure clause is useful when you want to make sure, for instance, that files are closed.

Note that, unlike the `else` clause, the `ensure` clause *is executed* before the `begin` or `rescue` clause returns a value. If the `ensure` clause has a `return` that will override the `return` value of any other clause!

Chapter 59: Command Line Apps

Section 59.1: How to write a command line tool to get the weather by zip code

This will be a relatively comprehensive tutorial of how to write a command line tool to print the weather from the zip code provided to the command line tool. The first step is to write the program in ruby to do this action. Let's start by writing a method `weather(zip_code)` (This method requires the `yahoo_weatherman` gem. If you do not have this gem you can install it by typing `gem install yahoo_weatherman` from the command line)

```
require 'yahoo_weatherman'

def weather(zip_code)
  client = Weatherman::Client.new
  client.lookup_by_location(zip_code).condition['temp']
end
```

We now have a very basic method that gives the weather when a zip code is provided to it. Now we need to make this into a command line tool. Very quickly let's go over how a command line tool is called from the shell and the associated variables. When a tool is called like this `tool argument other_argument`, in ruby there is a variable `ARGV` which is an array equal to `['argument', 'other_argument']`. Now let us implement this in our application

```
#!/usr/bin/ruby
require 'yahoo_weatherman'

def weather(zip_code)
  client = Weatherman::Client.new
  client.lookup_by_location(zip_code).condition['temp']
end

puts weather(ARGV[0])
```

Good! Now we have a command line application that can be run. Notice the *she-bang* line at the beginning of the file (`#!/usr/bin/ruby`). This allows the file to become an executable. We can save this file as `weather`. (**Note**: Do not save this as `weather.rb`, there is no need for the file extension and the she-bang tells whatever you need to tell that this is a ruby file). Now we can run these commands in the shell (do not type in the $).

```
$ chmod a+x weather
$ ./weather [ZIPCODE]
```

After testing that this works, we can now sym-link this to the `/usr/bin/local/` by running this command

```
$ sudo ln -s weather /usr/local/bin/weather
```

Now `weather` can be called on the command line no matter the directory you are in.

Chapter 60: IRB

Option	Details
-f	Suppress read of ~/.irbrc
-m	Bc mode (load mathn, fraction or matrix are available)
-d	Set $DEBUG to true (same as `ruby -d')
-r load-module	Same as `ruby -r'
-I path	Specify $LOAD_PATH directory
-U	Same as `ruby -U`
-E enc	Same as `ruby -E`
-w	Same as `ruby -w`
-W[level=2]	Same as `ruby -W`
--inspect	Use `inspect' for output (default except for bc mode)
--noinspect	Don't use inspect for output
--readline	Use Readline extension module
--noreadline	Don't use Readline extension module
--prompt prompt-mode	Switch prompt mode. Pre-defined prompt modes are `default`,simple', `xmp` andinf-ruby'
--inf-ruby-mode	Use prompt appropriate for inf-ruby-mode on emacs. Suppresses --readline.
--simple-prompt	Simple prompt mode
--noprompt	No prompt mode
--tracer	Display trace for each execution of commands.
--back-trace-limit n	Display backtrace top n and tail n. The default value is 16.
--irb_debug n	Set internal debug level to n (not for popular use)
-v, --version	Print the version of irb

IRB means "Interactive Ruby Shell". Basically it lets you execute ruby commands in real time (like the normal shell does). IRB is an indispensable tool when dealing with Ruby API. Works as classical rb script. Use it for short and easy commands. One of the nice IRB functions is that when you press tab while typing a method it will give you an advice to what you can use (This is not an IntelliSense)

Section 60.1: Starting an IRB session inside a Ruby script

As of Ruby 2.4.0, you can start an interactive IRB session inside any Ruby script using these lines:

```
require 'irb'
binding.irb
```

This will start an IBR REPL where you will have the expected value for `self` and you will be able to access all local variables and instance variables that are in scope. Type Ctrl+D or `quit` in order to resume your Ruby program.

This can be very useful for debugging.

Section 60.2: Basic Usage

IRB means "Interactive Ruby Shell", letting us execute ruby expressions from the standart input.

To start, type `irb` into your shell. You can write anything in Ruby, from simple expressions:

```
$ irb
2.1.4 :001 > 2+2
=> 4
```

to complex cases like methods:

```
2.1.4 :001> def method
2.1.4 :002?>   puts "Hello World"
2.1.4 :003?> end
=> :method
2.1.4 :004 > method
Hello World
=> nil
```

Chapter 61: ERB

ERB stands for Embedded Ruby, and is used to insert Ruby variables inside templates, e.g. HTML and YAML. ERB is a Ruby class that accepts text, and evaluates and replaces Ruby code surrounded by ERB markup.

Section 61.1: Parsing ERB

This example is filtered text from an `IRB` session.

```
=> require 'erb'
=> input = <<-HEREDOC
<ul>
<% (0..10).each do |i| %>
    <%# This is a comment %>
    <li><%= i %> is <%= i.even? ? 'even' : 'odd' %>.</li>
<% end %>
</ul>
HEREDOC

=> parser = ERB.new(input)
=> output = parser.result
=> print output
<ul>


    <li>0 is even.</li>


    <li>1 is odd.</li>


    <li>2 is even.</li>


    <li>3 is odd.</li>


    <li>4 is even.</li>


    <li>5 is odd.</li>


    <li>6 is even.</li>


    <li>7 is odd.</li>


    <li>8 is even.</li>


    <li>9 is odd.</li>


    <li>10 is even.</li>

</ul>
```

Chapter 62: Generate a random number

How to generate a random number in Ruby.

Section 62.1: 6 Sided die

```
# Roll a 6 sided die, rand(6) returns a number from 0 to 5 inclusive
dice_roll_result = 1 + rand(6)
```

Section 62.2: Generate a random number from a range (inclusive)

```
# ruby 1.92
lower_limit = 1
upper_limit = 6
Random.new.rand(lower_limit..upper_limit) # Change your range operator to suit your needs
```

Chapter 63: Getting started with Hanami

My mission here is to contribute with the community to help new people who wants to learn about this amazing framework - Hanami.

But how it is going to work?

Short and easygoing tutorials showing with examples about Hanami and following the next tutorials we will see how to test our application and build a simple REST API.

Let's start!

Section 63.1: About Hanami

Besides Hanami be a lightweight and fast framework one of the points that most call attention is the **Clean Architecture** concept where shows to us that the framework is not our application as Robert Martin said before.

Hanami arquitecture design offer to us the use of **Container**, in each Container we have our application independently of the framework. This means that we can grab our code and put it into a Rails framework for example.

Hanami is a MVC Framework?

The MVC's frameworks idea is to build one structure following the Model -> Controller -> View. Hanami follows the Model | Controller -> View -> Template. The result is an application more uncopled, following **SOLID** principles, and much cleaner.

- Important links.

Hanami http://hanamirb.org/

Robert Martin - Clean Arquitecture https://www.youtube.com/watch?v=WpkDN78P884

Clean Arquitecture https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

SOLID Principles http://practicingruby.com/articles/solid-design-principles

Section 63.2: How to install Hanami?

- **Step 1:** Installing the Hanami gem.

```
$ gem install hanami
```

- **Step 2**: Generate a new project setting RSpec as testing framework.

  Open up a command line or terminal. To generate a new hanami application, use hanami new followed by the name of your app and the rspec test param.

```
$ hanami new "myapp" --test=rspec
```

Obs. By default Hanami sets Minitest as testing framework.

This will create a hanami application called myapp in a myapp directory and install the gem dependencies that are already mentioned in Gemfile using bundle install.

To switch to this directory, use the cd command, which stands for change directory.

```
$ cd my_app
$ bundle install
```

The myapp directory has a number of auto-generated files and folders that make up the structure of a Hanami application. Following is a list of files and folders that are created by default:

- **Gemfile** defines our Rubygems dependencies (using Bundler).
- **Rakefile** describes our Rake tasks.
- **apps** contains one or more web applications compatible with Rack. Here we can find the first generated Hanami application called Web. It's the place where we find our controllers, views, routes and templates.
- **config** contains configuration files.
- **config.ru** is for Rack servers.
- **db** contains our database schema and migrations.
- **lib** contains our business logic and domain model, including entities and repositories.
- **public** will contain compiled static assets.
- **spec** contains our tests.

- **Important links.**

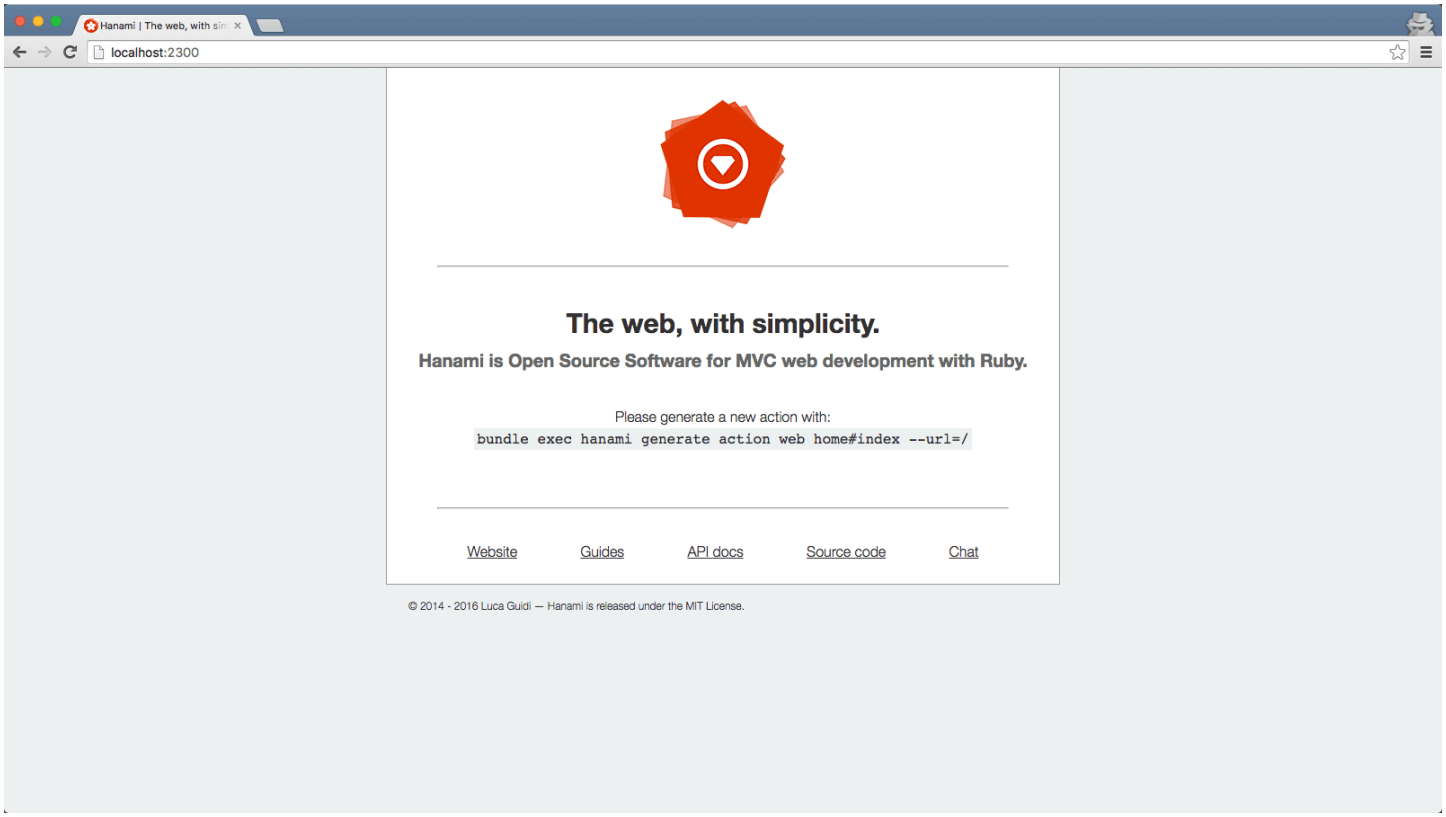
Hanami gem https://github.com/hanami/hanami

Hanami official Getting Started http://hanamirb.org/guides/getting-started/

Section 63.3: How to start the server?

- **Step 1:** To start the server just type the command bellow then you'll see the start page.

```
$ bundle exec hanami server
```


Hanami | The web, with sim
localhost:2300
The web, with simplicity.
Hanami is Open Source Software for MVC web development with Ruby.
Please generate a new action with:
bundle exec hanami generate action web home#index --url=/
Website
Guides
API docs
Source code
Chat
© 2014 - 2016 Luca Guidi — Hanami is released under the MIT License.

Chapter 64: OptionParser

OptionParser can be used for parsing command line options from `ARGV`.

Section 64.1: Mandatory and optional command line options

It's relatively easy to parse the command line by hand if you aren't looking for anything too complex:

```
# Naive error checking
abort('Usage: ' + $0 + ' site id ...') unless ARGV.length >= 2

# First item (site) is mandatory
site = ARGV.shift

ARGV.each do | id |
  # Do something interesting with each of the ids
end
```

But when your options start to get more complicated, you probably will need to use an option parser such as, well, OptionParser:

```
require 'optparse'

# The actual options will be stored in this hash
options = {}

# Set up the options you are looking for
optparse = OptionParser.new do |opts|
  opts.banner = "Usage: #{$0} -s NAME id ..."

  opts.on("-s", "--site NAME", "Site name") do |s|
    options[:site] = s
  end

  opts.on( '-h', '--help', 'Display this screen' ) do
    puts opts
    exit
  end
end

# The parse! method also removes any options it finds from ARGV.
optparse.parse!
```

There's also a non-destructive `parse`, but it's a lot less useful if you plan on using the remainder of what's in `ARGV`.

The OptionParser class doesn't have a way to enforce mandatory arguments (such as `--site` in this case). However you can do you own checking after running `parse!`:

```
# Slightly more sophisticated error checking
if options[:site].nil? or ARGV.length == 0
  abort(optparse.help)
end
```

For a more generic mandatory option handler, see this answer. In case it isn't clear, all options are optional unless you go out of your way to make them mandatory.

Section 64.2: Default values

With `OptionsParser`, it's really easy to set up default values. Just pre-populate the hash you store the options in:

```
options = {
  :directory => ENV['HOME']
}
```

When you define the parser, it will overwrite the default if a user provide a value:

```
OptionParser.new do |opts|
  opts.on("-d", "--directory HOME", "Directory to use") do |d|
    options[:directory] = d
  end
end
```

Section 64.3: Long descriptions

Sometimes your description can get rather long. For instance `irb -h` lists on argument that reads:

```
  --context-mode n  Set n[0-3] to method to create Binding Object,
                    when new workspace was created
```

It's not immediately clear how to support this. Most solutions require adjusting to make the indentation of the second and following lines align to the first. Fortunately, the `on` method supports multiple description lines by adding them as separate arguments:

```
  opts.on("--context-mode n",
          "Set n[0-3] to method to create Binding Object,",
          "when new workspace was created") do |n|
    optons[:context_mode] = n
  end
```

You can add as many description lines as you like to fully explain the option.

Chapter 65: Operating System or Shell commands

There are many ways to interact with the operating system. From within Ruby you can run shell/system commands or sub-processes.

Section 65.1: Recommended ways to execute shell code in Ruby:

Open3.popen3 or Open3.capture3:
Open3 actually just uses Ruby's spawn command, but gives you a much better API.

Open3.popen3

Popen3 runs in a sub-process and returns stdin, stdout, stderr and wait_thr.

```
require 'open3'
stdin, stdout, stderr, wait_thr = Open3.popen3("sleep 5s && ls")
puts "#{stdout.read} #{stderr.read} #{wait_thr.value.exitstatus}"
```

or

```
require 'open3'
cmd = 'git push heroku master'
Open3.popen3(cmd) do |stdin, stdout, stderr, wait_thr|
  puts "stdout is:" + stdout.read
  puts "stderr is:" + stderr.read
end
```

will output: **stdout is: stderr is:fatal: Not a git repository (or any of the parent directories): .git**

or

```
require 'open3'
cmd = 'ping www.google.com'
Open3.popen3(cmd) do |stdin, stdout, stderr, wait_thr|
  while line = stdout.gets
    puts line
  end
end
```

will output:

Pinging www.google.com [216.58.223.36] with 32 bytes of data:
Reply from 216.58.223.36: bytes=32 time=16ms TTL=54
Reply from 216.58.223.36: bytes=32 time=10ms TTL=54
Reply from 216.58.223.36: bytes=32 time=21ms TTL=54
Reply from 216.58.223.36: bytes=32 time=29ms TTL=54
Ping statistics for 216.58.223.36:
Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
Minimum = 10ms, Maximum = 29ms, Average = 19ms

Open3.capture3:

```
require 'open3'

stdout, stderr, status = Open3.capture3('my_funky_command', 'and', 'some', 'argumants')
if status.success?
  # command completed successfully, do some more stuff
else
  raise "An error occurred"
end
```

or

```
Open3.capture3('/some/binary with some args')
```

Not recommended though, due to additional overhead and the potential for shell injections.

If the command reads from stdin and you want to feed it some data:

```
Open3.capture3('my_funky_command', stdin_data: 'read from stdin')
```

Run the command with a different working directory, by using chdir:

```
Open3.capture3('my_funky_command', chdir: '/some/directory')
```

Section 65.2: Clasic ways to execute shell code in Ruby:

Exec:

```
exec 'echo "hello world"'
```

or

```
exec ('echo "hello world"')
```

The System Command:

```
system 'echo "hello world"'
```

Will output "hello world" in the command window.

or

```
system ('echo "hello world"')
```

The system command can return a true if the command was successful or nill when not.

```
result = system 'echo "hello world"'
puts result  # will return a true in the command window
```

The backticks (`):

`echo "hello world"` Will output "hello world" in the command window.

You can also catch the result.

```ruby
result = `echo "hello world"`
puts "We always code a " + result
```

IO.popen:

```ruby
# Will get and return the current date from the system
IO.popen("date") { |f| puts f.gets }
```

Chapter 66: C Extensions

Section 66.1: Your first extension

C extensions are comprised of two general pieces:

1. The C Code itself.
2. The extension configuration file.

To get started with your first extension put the following in a file named `extconf.rb`:

```
require 'mkmf'

create_makefile('hello_c')
```

A couple of things to point out:

First, the name `hello_c` is what the output of your compiled extension is going to be named. It will be what you use in conjunction with `require`.

Second, the `extconf.rb` file can actually be named anything, it's just traditionally what is used to build gems that have native code, the file that is actually going to compile the extension is the Makefile generated when running `ruby extconf.rb`. The default Makefile that is generated compiles all `.c` files in the current directory.

Put the following in a file named `hello.c` and run `ruby extconf.rb && make`

```
#include <stdio.h>
#include "ruby.h"

VALUE world(VALUE self) {
  printf("Hello World!\n");
  return Qnil;
}

// The initialization method for this module
void Init_hello_c() {
  VALUE HelloC = rb_define_module("HelloC");
  rb_define_singleton_method(HelloC, "world", world, 0);
}
```

A breakdown of the code:

The name `Init_hello_c` must match the name defined in your `extconf.rb` file, otherwise when dynamically loading the extension, Ruby won't be able to find the symbol to bootstrap your extension.

The call to `rb_define_module` is creating a Ruby module named `HelloC` which we're going to namespace our C functions under.

Finally, the call to `rb_define_singleton_method` makes a module level method tied directly to the `HelloC` module which we can invoke from ruby with `HelloC.world`.

After having compiled the extension with the call to `make` we can run the code in our C extension.

Fire up a console!

```
irb(main):001:0> require './hello_c'
```

```
=> true
irb(main):002:0> HelloC.world
Hello World!
=> nil
```

Section 66.2: Working with C Structs

In order to be able to work with C structs as Ruby objects, you need to wrap them with calls to `Data_Wrap_Struct` and `Data_Get_Struct`.

`Data_Wrap_Struct` wraps a C data structure in a Ruby object. It takes a pointer to your data structure, along with a few pointers to callback functions, and returns a VALUE. The `Data_Get_Struct` macro takes that VALUE and gives you back a pointer to your C data structure.

Here's a simple example:

```
#include <stdio.h>
#include <ruby.h>

typedef struct example_struct {
  char *name;
} example_struct;

void example_struct_free(example_struct * self) {
  if (self->name != NULL) {
    free(self->name);
  }
  ruby_xfree(self);
}

static VALUE rb_example_struct_alloc(VALUE klass) {
  return Data_Wrap_Struct(klass, NULL, example_struct_free, ruby_xmalloc(sizeof(example_struct)));
}

static VALUE rb_example_struct_init(VALUE self, VALUE name) {
  example_struct* p;

  Check_Type(name, T_STRING);

  Data_Get_Struct(self, example_struct, p);
  p->name = (char *)malloc(RSTRING_LEN(name) + 1);
  memcpy(p->name, StringValuePtr(name), RSTRING_LEN(name) + 1);

  return self;
}

static VALUE rb_example_struct_name(VALUE self) {
  example_struct* p;
  Data_Get_Struct(self, example_struct, p);

  printf("%s\n", p->name);

  return Qnil;
}

void Init_example()
{
  VALUE mExample = rb_define_module("Example");
  VALUE cStruct = rb_define_class_under(mExample, "Struct", rb_cObject);
```

```
  rb_define_alloc_func(cStruct, rb_example_struct_alloc);
  rb_define_method(cStruct, "initialize", rb_example_struct_init, 1);
  rb_define_method(cStruct, "name", rb_example_struct_name, 0);
}
```

And the `extconf.rb`:

```
require 'mkmf'

create_makefile('example')
```

After compiling the extension:

```
irb(main):001:0> require './example'
=> true
irb(main):002:0> test_struct = Example::Struct.new("Test Struct")
=> #<Example::Struct:0x007fc741965068>
irb(main):003:0> test_struct.name
Test Struct
=> nil
```

Section 66.3: Writing Inline C - RubyInLine

RubyInline is a framework that lets you embed other languages inside your Ruby code. It defines the Module# inline method, which returns a builder object. You pass the builder a string containing code written in a language other than Ruby, and the builder transforms it into something that you can call from Ruby.

When given C or C++ code (the two languages supported in the default RubyInline install), the builder objects writes a small extension to disk, compiles it, and loads it. You don't have to deal with the compilation yourself, but you can see the generated code and compiled extensions in the .ruby_inline subdirectory of your home directory.

Embed C code right in your Ruby program:

- RubyInline (available as the rubyinline gem) create an extension automatically

> RubyInline won't work from within irb

```
#!/usr/bin/ruby -w
    # copy.rb
    require 'rubygems'
    require 'inline'

    class Copier
    inline do |builder|
      builder.c <<END
    void copy_file(const char *source, const char *dest)
    {
      FILE *source_f = fopen(source, "r");
      if (!source_f)
      {
        rb_raise(rb_eIOError, "Could not open source : '%s'", source);
      }

      FILE *dest_f = fopen(dest, "w+");
      if (!dest_f)
      {
```

```
        rb_raise(rb_eIOError, "Could not open destination : '%s'", dest);
      }

      char buffer[1024];

      int nread = fread(buffer, 1, 1024, source_f);
      while (nread > 0)
      {
        fwrite(buffer, 1, nread, dest_f);
        nread = fread(buffer, 1, 1024, source_f);
      }
    }
    END
     end
    end
```

C function `copy_file` now exists as an instance method of `Copier`:

```
open('source.txt', 'w') { |f| f << 'Some text.' }
Copier.new.copy_file('source.txt', 'dest.txt')
puts open('dest.txt') { |f| f.read }
```

Chapter 67: Debugging

Section 67.1: Stepping through code with Pry and Byebug

First, you need to install `pry-byebug` gem. Run this command:

```
$ gem install pry-byebug
```

Add this line at the top of your `.rb` file:

```
require 'pry-byebug'
```

Then insert this line wherever you want a breakpoint:

```
binding.pry
```

A `hello.rb` example:

```
require 'pry-byebug'

def hello_world
  puts "Hello"
  binding.pry # break point here
  puts "World"
end
```

When you run the `hello.rb` file, the program will pause at that line. You can then step through your code with the `step` command. Type a variable's name to learn its value. Exit the debugger with `exit-program` or `!!!`.

Chapter 68: Ruby Version Manager

Section 68.1: How to create gemset

To create a gemset we need to create a `.rvmrc` file.

Syntax:

```
$ rvm --rvmrc --create <ruby-version>@<gemsetname>
```

Example:

```
$ rvm --rvmrc --create ruby-2.2.2@myblog
```

The above line will create a `.rvmrc` file in the root directory of the app.

To get the list of available gemsets, use the following command:

```
$ rvm list gemsets
```

Section 68.2: Installing Ruby with RVM

The *Ruby Version Manager* is a command line tool to simply install and manage different versions of Ruby.

- `rvm istall 2.3.1` for example installs Ruby version 2.3.1 on your machine.
- With `rvm list` you can see which versions are installed and which is actually set for use.

```
user@dev:~$ rvm list

rvm rubies

=* ruby-2.3.1 [ x86_64 ]

# => - current
# =* - current && default
#  * - default
```

- With `rvm use 2.3.0` you can change between installed versions.

Appendix A: Installation

Section A.1: Installing Ruby macOS

So the good news is that Apple kindly includes a Ruby interpreter. Unfortunately, it tends not to be a recent version:

```
$ /usr/bin/ruby -v
ruby 2.0.0p648 (2015-12-16 revision 53162) [universal.x86_64-darwin16]
```

If you have Homebrew installed, you can get the latest Ruby with:

```
$ brew install ruby

$ /usr/local/bin/ruby -v
ruby 2.4.1p111 (2017-03-22 revision 58053) [x86_64-darwin16]
```

(It's likely you'll see a more recent version if you try this.)

In order to pick up the brewed version without using the full path, you'll want to add `/usr/local/bin` to the start of your `$PATH` environment variable:

```
export PATH=/usr/local/bin:$PATH
```

Adding that line to `~/.bash_profile` ensures that you will get this version after you restart your system:

```
$ type ruby
ruby is /usr/local/bin/ruby
```

Homebrew will install `gem` for installing Gems. It's also possible to build from the source if you need that. Homebrew also includes that option:

```
$ brew install ruby --build-from-source
```

Section A.2: Gems

In this example we will use 'nokogiri' as an example gem. 'nokogiri' can later on be replaced by any other gem name.

To work with gems we use a command line tool called `gem` followed by an option like `install` or `update` and then names of the gems we want to install, but that is not all.

Install gems:

```
$> gem install nokogiri
```

But that is not the only thing we need. We can also specify version, source from which to install or search for gems. Lets start with some basic use cases (UC) and you can later on post request for an update.

Listing all the installed gems:

```
$> gem list
```

Uninstalling gems:

```
$> gem uninstall nokogiri
```

If we have more version of the nokogiri gem we will be prompted to specify which one we want to uninstall. We will get a list that is ordered and numbered and we just write the number.

Updating gems

```
$> gem update nokogiri
```

or if we want to update them all

```
$> gem update
```

Comman `gem` has many more usages and options to be explored. For more please turn to the official documentation. If something is not clear post a request and I will add it.

Section A.3: Linux - Compiling from source

`This way you will get the newest ruby but it has its downsides. Doing it like this ruby will not be managed by any application.

!! Remember to chagne the version so it coresponds with your !!

1. you need to download a tarball find a link on an official website (https://www.ruby-lang.org/en/downloads/)
2. Extract the tarball
3. Install

```
$> wget https://cache.ruby-lang.org/pub/ruby/2.3/ruby-2.3.3.tar.gz
$> tar -xvzf ruby-2.3.3.tar.gz
$> cd ruby-2.3.3
$> ./configure
$> make
$> sudo make install
```

This will install ruby into `/usr/local`. If you are not happy with this location you can pass an argument to the `./configure --prefix=DIR` where `DIR` is the directory you want to install ruby to.

Section A.4: Linux—Installation using a package manager

Probably the easiest choice, but beware, the version is not always the newest one. Just open up terminal and type (depending on your distribution)

in Debian or Ubuntu using apt

```
$> sudo apt install ruby
```

in CentOS, openSUSE or Fedora

```
$> sudo yum install ruby
```

You can use the `-y` option so you are not prompted to agree with the installation but in my opinion it is a good practice to always check what is the package manager trying to install.

Section A.5: Windows - Installation using installer

Probably the easies way to set up ruby on windows is to go to http://rubyinstaller.org/ and from there donwload an executable that you will install.

You don't have to set almost anything, but there will be one important window. It will have a check box saying *Add ruby executable to your PATH*. Confirm that it is **checked**, if not check it or else you won't be able to run ruby and will have to set the PATH variable on your own.

Then just go next until it installs and thats that.

Section A.6: Linux - troubleshooting gem install

First UC in the example **Gems** `$> gem install nokogiri` can have a problem installing gems because we don't have the permissions for it. This can be sorted out in more then just one way.

First UC solution a:

U can use `sudo`. This will install the gem for all the users. This method should be frowned upon. This should be used only with the gem you know will be usable by all the users. Usualy in real life you don't want some user having access to `sudo`.

```
$> sudo gem install nokogiri
```

First UC solution b

U can use the option `--user-install` which installs the gems into your users gem folder (usualy at `~/.gem`)

```
&> gem install nokogiri --user-install
```

First UC solution c

U can set GEM_HOME and GEM_PATH wich then will make command `gem install` install all the gems to a folder which you specify. I can give you an example of that (the usual way)

- First of all you need to open .bashrc. Use nano or your favorite text editor.

```
$> nano ~/.bashrc
```

- Then at the end of this file write

```
export GEM_HOME=$HOME/.gem
export GEM_PATH=$HOME/.gem
```

- Now you will need to restart terminal or write `. ~/.bashrc` to re-load the configuration. This will enable you to use `gem isntall nokogiri` and it will install those gems in the folder you specified.

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

Adam Sanderson	Chapter 18
Addison	Chapter 28
AJ Gregory	Chapter 11
Ajedi32	Chapter 9
alebruck	Chapter 9
Ali MasudianPour	Chapter 31
Alu	Chapters 68 and 40
amingilani	Chapter 61
Andrea Mazzarella	Chapter 9
Andrew	Chapter 55
angelparras	Chapter 17
Anthony Staunton	Chapter 44
Arman Jon Villalobos	Chapter 19
ArtOfCode	Chapter 3
Artur Tsuda	Chapters 15 and 27
Arun Kumar M	Chapter 15
Atul Khanduri	Chapter 19
Austin Vern Songer	Chapters 33, 47 and 66
Automatico	Chapter 9
bang bang	Chapters 14 and 31
br3nt	Chapters 9, 25, 20, 26, 17, 11 and 18
C dot StrifeVII	Chapters 19 and 49
CalmBit	Chapter 1
Charan Kumar Borra	Chapter 21
Charlie Egan	Chapters 19 and 11
Chris	Chapter 21
Christoph Petschnig	Chapter 19
Christopher Oezbek	Chapter 19
coreyward	Chapter 20
D	Chapters 9 and 17
daniero	Chapters 9 and 17
DarKy	Chapter 17
Darpan Chhatravala	Chapter 1
David Grayson	Chapters 1, 9, 19, 17, 11 and 60
David Ljung Madison	Chapter 18
davidhu2000	Chapters 9, 25 and 11
DawnPaladin	Chapters 1, 9, 34 and 67
Dimitry_N	Chapter 17
Divya Sharma	Chapter 31
divyum	Chapter 19
djaszczurowski	Chapter 31
Doodad	Chapter 29
Dorian	Chapter 37
Elenian	Chapters 25, 17 and 28
Eli Sadoff	Chapters 9, 20, 14, 21 and 59
engineersmnky	Chapter 20
Engr. Hasanuzzaman Sumon	Chapters 6 and 36
equivalent8	Chapter 41

Francesco Boffa	Chapter 10
Francesco Lupo Renzi	Chapters 9 and 17
G. Allen Morris III	Chapter 19
Gaelan	Chapter 26
Geoffroy	Chapter 45
gorn	Chapter 19
Hardik Kanjariya	Chapter 68
iGbanam	Chapter 9
iltempo	Chapter 19
Inanc Gumus	Chapter 44
iturgeon	Chapter 22
Jasper	Chapter 20
Jeweller	Chapter 19
JoeyB	Chapters 25 and 17
Jon Ericson	Chapter 28
Jon Wood	Chapter 2
Jonathan	Chapter 3
jose_castro_arnaud	Chapter 17
joshaidan	Chapter 9
Justin Chadwell	Chapter 25
kamaradclimber	Chapter 22
Kathryn	Chapters 19, 17, 11, 18, 43, 38, 69, 64 and 39
Katsuhiko Yoshida	Chapter 9
Kirti Thorat	Chapter 26
kleaver	Chapter 19
knut	Chapters 1 and 9
Koraktor	Chapter 19
Kris	Chapter 19
Lahiru	Chapter 17
Lomefin	Chapter 22
Lucas Costa	Chapters 1, 9, 19, 11, 22, 31, 5 and 13
Lukas Baliak	Chapters 9, 19, 20 and 22
lwassink	Chapter 9
Lynn	Chapter 35
mahatmanich	Chapter 18
manasouza	Chapter 42
Marc	Chapter 20
Martin Velez	Chapters 1, 25, 19 and 29
Masa Sakano	Chapter 9
Matheus Moreira	Chapters 4, 45, 23, 48, 49, 50 and 51
Mauricio Junior	Chapter 63
max pleaner	Chapters 29 and 57
Maxim Fedotov	Chapters 33 and 60
Maxim Pontyushenko	Chapter 21
meagar	Chapters 2, 9, 19, 20, 17 and 11
MegaTom	Chapters 25, 26, 33, 21, 35, 45, 13 and 56
meta	Chapter 49
Mhmd	Chapters 2, 9, 25, 19 and 20
Michael Gaskill	Chapter 17
Michael Kuhinica	Chapters 19 and 22
Mike H	Chapter 9
Milo P	Chapter 29
mlabarca	Chapter 26
moertel	Chapters 19, 17, 22 and 33
mrcasals	Chapter 35

mrlee	Chapter 19
MrTheWalrus	Chapter 9
mudasobwa	Chapter 28
Muhammad Abdullah	Chapters 11 and 22
MZaragoza	Chapter 19
Nakilon	Chapter 2
NateSHolland	Chapter 44
NateW	Chapter 11
ndn	Chapters 17, 14, 21 and 16
Neha Chopra	Chapters 24 and 30
neontapir	Chapter 19
New Alexandria	Chapter 19
Nic Nilov	Chapter 19
Nick Roz	Chapters 9, 25, 19, 17, 11, 21 and 28
Ninigi	Chapter 4
Nuno Silva	Chapter 29
nus	Chapters 19, 20, 26, 18, 32, 3, 35 and 45
ogirginc	Chapter 67
Old Pro	Chapter 19
Owen	Chapter 15
Ozgur Akyazi	Chapter 21
Pablo Torrecilla	Chapters 9 and 17
paradoja	Chapter 37
peter	Chapter 28
philomory	Chapter 25
photoionized	Chapter 66
Phrogz	Chapters 3, 50 and 56
pjam	Chapter 19
pjrebsch	Chapter 15
PJSCopeland	Chapter 19
Pooyan Khosravi	Chapters 45, 46, 50 and 52
Pragash	Chapter 26
QPaysTaxes	Chapter 20
Rahul Singh	Chapters 34 and 8
Redithion	Chapters 18, 14, 7 and 12
Richard Hamilton	Chapters 9, 11 and 14
Roan Fourie	Chapter 65
Robert Columbia	Chapter 22
russt	Chapter 17
Sagar Pandya	Chapter 9
SajithP	Chapters 21 and 38
Saroj Sasmal	Chapter 9
Saša Zejnilović	Chapters 44 and 69
Scudelletti	Chapter 28
Sean Redmond	Chapter 58
Shadoath	Chapter 9
Shelvacu	Chapter 28
Sid	Chapters 19 and 11
SidOfc	Chapter 44
Simon Soriano	Chapter 49
Simone Carletti	Chapters 1, 25, 19, 11, 15, 22, 18, 14 and 53
snonov	Chapter 5
Sourabh Upadhyay	Chapter 49
spencer.sm	Chapter 25
squadette	Chapter 9

Steve	Chapters 1 and 17
stevendaniels	Chapters 25, 19, 13 and 58
suhao399	Chapter 36
Surya	Chapter 33
thesecretmaster	Chapter 42
Tom Harrison Jr	Chapters 27 and 3
Tom Lord	Chapters 9, 19, 17, 14 and 44
Tot Zam	Chapter 1
Umang Raghuvanshi	Chapter 54
user1213904	Chapter 31
user1489580	Chapter 44
user1821961	Chapter 62
user2367593	Chapter 6
Vasfed	Chapters 26, 11 and 35
Ven	Chapters 19 and 17
vgoff	Chapter 17
Vidur	Chapter 43
Vishnu Y S	Chapter 1
wirefox	Chapters 19 and 14
wjordan	Chapter 11
xavdid	Chapter 57
Yonatha Almeida	Chapter 34
Yule	Chapters 17 and 27
Zaz	Chapters 18 and 47

You may also like