Ruby 2.5 Information and Documentation

OCTOBER, 2018

wlharvey4

Published by:

wlharvey4
Address Line 1
Address Line 2
etc.
Email: wlharvey4@emac.com
URL: http://www.example.com/

This is Edition 0.1e of *Ruby 2.5 Information and Documentation*.

Table of Contents

Preface

Text here.

Intended Audience

Text here.

What Is Covered

Text and chapter by chapter description here.

Typographical Conventions

This book is written in an enhanced version of Texinfo, the GNU documentation formatting language. A single Texinfo source file is used to produce both the printed and online versions of a program's documentation. Because of this, the typographical conventions are slightly different than in other books you may have read.

Examples you would type at the command-line are preceded by the common shell primary and secondary prompts, '`$`' and '`>`'. Input that you type is shown *`like this`*. Output from the command is preceded by the glyph "⊣". This typically represents the command's standard output. Error messages, and other output on the command's standard error, are preceded by the glyph " error ". For example:

```
$ echo hi on stdout
⊣ hi on stdout
$ echo hello on stderr 1>&2
error  hello on stderr
```

In the text, command names appear in `this font`, while code segments appear in the same font and quoted, '`like this`'. Options look like this: `-f`. Some things are emphasized *like this*, and if a point needs to be made strongly, it is done **like this**. The first occurrence of a new term is usually its *definition* and appears in the same font as the previous occurrence of "definition" in this sentence. Finally, file names are indicated like this: `/path/to/our/file`.

Acknowledgements

1 Introduction

Ruby is . . .

A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.

2 Documentation

Here you will find pointers to manuals, tutorials and references that will come in handy when you feel like coding in Ruby.

2.1 Installing Ruby

Installation Methods

There are several ways to install Ruby:

- **Package Manager**: When you are on a UNIX-like operating system, using your systems package manager is the easiest way of getting started. However, the packaged Ruby version usually is not the newest one.
- **Installers**: can be used to install a specific or multiple Ruby versions. There is also an installer for Windows.
- **Managers** help you to switch between multiple Ruby installations on your system.
- **Source**: And finally, you can also build Ruby from source.

The following overview lists available installation methods for different needs and platforms.

2.1.1 Package Management Systems

If you cannot compile your own Ruby, and you do not want to use a third-party tool, you can use your systems package manager to install Ruby.

Certain members in the Ruby community feel very strongly that you should never use a package manager to install Ruby and that you should use tools instead. While the full list of pros and cons is outside of the scope of this page, the most basic reason is that most package managers have older versions of Ruby in their official repositories. If you would like to use the newest Ruby, make sure you use the correct package name, or use the tools described further below instead.

2.1.1.1 Homebrew (OS X)

Homebrew

On macOS (High) Sierra and OS X El Capitan, Ruby 2.0 is included.

Many people on OS X use Homebrew as a package manager. It is really easy to get a newer version of Ruby using Homebrew:

```
$ brew install ruby
```

This should install the latest Ruby version.

2.1.2 Installers

If the version of Ruby provided by your system or package manager is out of date, a newer one can be installed using a third-party installer. Some of them also allow you to install multiple versions on the same system; associated managers can help to switch between the different Rubies. If you are planning to use RVM as a version manager you do not need a separate installer, it comes with its own.

2.1.2.1 `ruby-build`

`ruby-build`

`rbenv`

`ruby-build` is a plugin for `rbenv` (see Section 2.1.3.2 "`rbenv`", page 4, that allows you to compile and install different versions of Ruby into arbitrary directories. `ruby-build` can also be used as a standalone program without `rbenv`. It is available for OS X, Linux, and other UNIX-like operating systems.

2.1.2.2 `ruby-install`

`ruby-install` version manager `chruby` version switcher

`ruby-install`

`chruby`

`ruby-install` allows you to compile and install different versions of Ruby into arbitrary directories. There is also a sibling, `chruby` (see Section 2.1.3.1 "chruby", page 4), which handles switching between Ruby versions. It is available for OS X, Linux, and other UNIX-like operating systems.

2.1.3 Managers

Many Rubyists use Ruby managers to manage multiple Rubies. They confer various advantages but are not officially supported. Their respective communities are very helpful, however.

2.1.3.1 chruby

`chruby` allows you to switch between multiple Rubies. `chruby` can manage Rubies installed by `ruby-install` (see Section 2.1.2.2 "`ruby-install`", page 4) or even built from source.

2.1.3.2 `rbenv`

`rbenv`

`ruby-build`

`rbenv` allows you to manage multiple installations of Ruby. It does not support installing Ruby, but there is a popular plugin named `ruby-build` (see Section 2.1.2.1 "`ruby-build`", page 4) to install Ruby. Both tools are available for OS X, Linux, or other UNIX-like operating systems.

2.1.3.3 RVM ("Ruby Version Manager")

RVM

RVM allows you to install and manage multiple installations of Ruby on your system. It can also manage different gemsets. It is available for OS X, Linux, or other UNIX-like operating systems.

2.1.3.4 `uru`

Uru

Uru is a lightweight, multi-platform command line tool that helps you to use multiple Rubies on OS X, Linux, or Windows systems.

2.1.4 Building From Source

Ruby 2.5.1

Ruby Github

Of course, you can install Ruby from source. Download and unpack a tarball, then just do this:

```
$ ./configure
$ make
$ sudo make install
```

By default, this will install Ruby into `/usr/local`. To change, pass the `--prefix=DIR` option to the `./configure` script.

Using the third-party tools or package managers might be a better idea, though, because the installed Ruby wont be managed by any tools.

Installing from the source code is a great solution for when you are comfortable enough with your platform and perhaps need specific settings for your environment. Its also a good solution in the event that there are no other premade packages for your platform.

2.1.4.1 Releases Page

Releases Page

For more information about specific releases, particularly older releases or previews, see the Releases page.

This page lists individual Ruby releases.

Ruby 2.5.1 Released

ruby-2.1.5.tar.gz

Posted by naruse on 28 Mar 2018

This release includes some bug fixes and some security fixes.

- CVE-2017-17742: HTTP response splitting in WEBrick
- CVE-2018-6914: Unintentional file and directory creation with directory traversal in tempfile and tmpdir
- CVE-2018-8777: DoS by large request in WEBrick
- CVE-2018-8778: Buffer under-read in String#unpack
- CVE-2018-8779: Unintentional socket creation by poisoned NUL byte in UNIXServer and UNIXSocket
- CVE-2018-8780: Unintentional directory traversal by poisoned NUL byte in Dir
- Multiple vulnerabilities in RubyGems

2.1.4.2 Branches Page

Branches Page

Information about the current maintenance status of the various Ruby branches can be found on the Branches page.

This page lists the current maintenance status of the various Ruby branches. This is a preliminary list of Ruby branches and their maintenance status. The shown dates are inferred from the English versions of release posts or EOL announcements.

The Ruby branches or release series are categorized below into the following phases:

- normal maintenance (bug fix): Branch receives general bug fixes and security fixes.
- security maintenance (security fix): Only security fixes are backported to this branch.
- eol (end-of-life): Branch is not supported by the ruby-core team any longer and does not receive any fixes. No further patch release will be released.
- preview: Only previews or release candidates have been released for this branch so far.

Ruby 2.6

https://cache.ruby-lang.org/pub/ruby/2.6/ruby-2.6.0-preview2.tar.gz

ruby-2.6.0-preview2
status: preview
release date:

Ruby 2.5

https://cache.ruby-lang.org/pub/ruby/2.5/ruby-2.5.1.tar.gz

status: normal maintenance
release date: 2017-12-25

Ruby 2.4

https://cache.ruby-lang.org/pub/ruby/2.4/ruby-2.4.4.tar.gz

status: normal maintenance
release date: 2016-12-25

Ruby 2.3

https://cache.ruby-lang.org/pub/ruby/2.3/ruby-2.3.7.tar.gz

status: security maintenance
release date: 2015-12-25
EOL date: scheduled for 2019-03-31

Ruby 2.2

status: eol
release date: 2014-12-25
EOL date: 2018-03-31

2.1.4.3 Ruby Issue Tracking System

Bugs

How to report a bug

How To Report

Ruby Trunk

Ruby Trunk

All Issues

2.2 Developing Ruby

Ruby Core

Now is a fantastic time to follow Rubys development. With the increased attention Ruby has received in the past few years, theres a growing need for good talent to help enhance Ruby and document its parts. So, where do you start?

Ruby Core

The topics related to Ruby development covered here are:

- "Developing Ruby", page 7,
- "Developing Ruby", page 7,
- "Patch by Patch", page 7,
- Rules for Core Developers

Using Subversion to Track Ruby Development

Getting the latest Ruby source code is a matter of an anonymous checkout from the Subversion repository. From your command line:

```
$ svn co https://svn.ruby-lang.org/repos/ruby/trunk ruby
```

The `ruby` directory will now contain the latest source code for the development version of Ruby (`ruby-trunk`). Currently patches applied to the trunk are backported to the stable 2.5, 2.4, and 2.3 branches (see below).

If youd like to follow patching of Ruby 2.5, you should use the ruby_2_5 branch when checking out:

```
$ svn co https://svn.ruby-lang.org/repos/ruby/branches/ruby_2_5
```

This will check out the respective development tree into a `ruby_2_5` directory. Developers working on the maintenance branches are expected to migrate their changes to Rubys trunk, so often the branches are very similar, with the exception of improvements made by Matz and Nobu to the language itself.

If you prefer, you may browse Rubys Subversion repository via the web.

How to Use Git With the Main Ruby Repository

Those who prefer to use Git over Subversion can find instructions with the mirror on GitHub, both for those with commit access and everybody else.

Improving Ruby, Patch by Patch

The core team maintains an issue tracker for submitting patches and bug reports to Matz and the gang. These reports also get submitted to the Ruby-Core mailing list for discussion, so you can be sure your request wont go unnoticed. You can also send your patches straight to the mailing list. Either way, you are encouraged to take part in the discussion that ensues.

Please look over the Patch Writers Guide for some tips, straight from Matz, on how to get your patches considered.

Steps for Building a Patch

2.3 Getting Started

2.3.1 Try Ruby!

Try Ruby!

An interactive tutorial that lets you try out Ruby right in your browser. This 15-minute tutorial is aimed at beginners who want to get a feeling of the language.

2.3.2 Official FAQ

The official frequently asked questions.

FAQ

This document contains Frequently Asked Questions about Ruby with answers.

This FAQ is based on The Ruby Language FAQ originally compiled by Shugo Maeda and translated into English by Kentaro Goto. Thanks to Zachary Scott and Marcus Stollsteimer for incorporating the FAQ into the site and for a major overhaul of the content.

- General questions
- How does Ruby stack up against?
- Installing Ruby
- Variables, constants, and arguments
- Section 2.3.2.1 “FAQ Iterators”, page 8,
- Section 2.3.2.2 “FAQ Syntax”, page 10,
- Methods
- Classes and modules
- Built-in libraries
- Extension library
- Other features

2.3.2.1 FAQ Iterators

What is an iterator?

An iterator is a method which accepts a block or a `Proc` object. In the source file, the block is placed immediately after the invocation of the method. Iterators are used to produce user-defined control structures — especially loops.

Lets look at an example to see how this works. Iterators are often used to repeat the same action on each element of a collection, like this:

```
data = [1, 2, 3]
data.each do |i|
  puts i
end
```

The `each` method of the array `data` is passed the `do ... end` block, and executes it repeatedly. On each call, the block is passed successive elements of the array.

You can define blocks with `{ ... }` in place of `do ... end`.

```
data = [1, 2, 3]
data.each { |i|
  puts i
}
```

This code has the same meaning as the last example. However, in some cases, precedence issues cause `do ... end` and `{ ... }` to act differently.

```
foobar a, b do ... end # foobar is the iterator.
foobar a, b { ... } # b is the iterator.
```

This is because `{ ... }` binds more tightly to the preceding expression than does a `do ... end` block. The first example is equivalent to '`foobar(a, b) do ... end`', while the second is '`foobar(a, b { ... })`'.

How can I pass a block to an iterator?

You simply place the block after the iterator call. You can also pass a `Proc` object by prepending `&` to the variable or constant name that refers to the `Proc`.

How is a block used in an iterator?

This section or parts of it might be out-dated or in need of confirmation.

There are three ways to execute a block from an iterator method:

1. the `yield` control structure;

   The `yield` statement calls the block, optionally passing it one or more arguments.

   ```
   def my_iterator
     yield 1, 2
   end

   my_iterator {|a, b| puts a, b }
   ```

2. calling a `Proc` argument (made from a block) with `call`;

   If a method definition has a block argument (the last formal parameter has an ampersand (`&`) prepended), it will receive the attached block, converted to a `Proc` object. This may be called using `prc.call(args)`.

   ```
   def my_iterator(&b)
     b.call(1, 2)
   end

   my_iterator {|a, b| puts a, b }
   ```

   and

3. using `Proc.new` followed by a `call`.

   `Proc.new` (or the equivalent `proc` or `lambda` calls), when used in an iterator definition, takes the block which is given to the method as its argument and generates a procedure object from it. (`proc` and `lambda` are effectively synonyms.)

[Update needed: `lambda` *behaves in a slightly different way and produces a warning* '`tried to create Proc object without a block`'*.]*

```
def my_iterator
  Proc.new.call(3, 4)
  proc.call(5, 6)
  lambda.call(7, 8)
end

my_iterator {|a, b| puts a, b }
```

Perhaps surprisingly, `Proc.new` and friends do not in any sense consume the block attached to the method — each call to `Proc.new` generates a new procedure object out of the same block.

You can tell if there is a block associated with a method by calling `block_given?`.

What does Proc.new without a block do?

`Proc.new` without a block cannot generate a procedure object and an error occurs. In a method definition, however, `Proc.new` without a block implies the existence of a block at the time the method is called, and so no error will occur.

How can I run iterators in parallel?

See `http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/5252`

2.3.2.2 FAQ Syntax

List of FAQ items:

"FAQ Syntax", page 10,
"FAQ Syntax", page 11,
"FAQ Syntax", page 11,
"FAQ Syntax", page 11,
"FAQ Syntax", page 11,
"FAQ Syntax", page 12,
"FAQ Syntax", page 12,
"FAQ Syntax", page 12,
"FAQ Syntax", page 12,
"FAQ Syntax", page 13,
"FAQ Syntax", page 13,
"FAQ Syntax", page 13,
"FAQ Syntax", page 13,
"FAQ Syntax", page 13,
"FAQ Syntax", page 14,

What is the difference between an immediate value and a reference?

`Fixnum`, `true`, `nil`, and `false` are implemented as *immediate values*. With immediate values, variables hold the objects themselves, rather than references to them.

Singleton methods cannot be defined for such objects. Two `Fixnums` of the same value always represent the same object instance, so (for example) instance variables for the `Fixnum` with the value 1 are shared between all the 1's in the system. This makes it impossible to define a singleton method for just one of these.

What is the difference between `nil` and `false`?

First the similarity: `nil` and `false` are the only two objects that evaluate to `false` in a boolean context. (In other words: they are the only "falsy" values; all other objects are "truthy".)

However, `nil` and `false` are instances of different classes (`NilClass` and `FalseClass`), and have different behavior elsewhere.

We recommend that *predicate methods* (those whose name ends with a question mark) return `true` or `false`. Other methods that need to indicate failure should return `nil`.

The Empty String

An empty string (`""`) returns `true` in a conditional expression! In Perl, its `false`. Its very simple: in Ruby, only `nil` and `false` are `false` in conditional contexts.

You can use `empty?`, compare the string to `""`, or compare the strings size or length to 0 to find out if a string is empty.

A Symbol Object

What does `:name` mean?

A colon followed by a name generates a *Symbol object* which corresponds one-to-one with the identifier. During the duration of a program's execution the same Symbol object will be created for a given name or string. Symbols can also be created with `"name".intern` or `"name".to_sym`.

Symbol objects can represent identifiers for methods, variables, and so on. Some methods, like `define_method`, `method_missing`, or `trace_var`, require a symbol. Other methods, e.g. `attr_accessor`, `send`, or `autoload`, also accept a string.

Due to the fact that they are created only once, Symbols are often used as hash keys. String hash keys would create a new object for every single use, thereby causing some memory overhead. There is even a special syntax for symbol hash keys:

```
person_1 = { :name => "John", :age => 42 }
person_2 = { name: "Jane", age: 24 }        # alternate syntax
```

Symbols can also be used as enumeration values or to assign unique values to constants:

```
status = :open  # :closed, ...

NORTH = :NORTH
SOUTH = :SOUTH
```

How can I access the value of a symbol?

To get the value of the variable corresponding to a symbol, you can use `symbol.to_s` or `"#{symbol}"` to get the name of the variable, and then `eval` that in the scope of the symbol to get the variables contents:

```
a = "This is the content of 'a'"
b = eval("#{:a}")
a.object_id == b.object_id  # => true
```

You can also use:

```
b = binding.local_variable_get(:a)
```

If your symbol corresponds to the name of a method, you can use `send`:

```
class Demo
  def hello
    "Hello, world"
  end
end

demo = Demo.new
demo.send(:hello)
```

Or you can use `Object#method` to return a corresponding `Method` object, which you may then call:

```
m = demo.method(:hello)  # => #<Method: Demo#hello>
m.call                   # => "Hello, world"
```

Is `loop` a control structure?

Although `loop` looks like a control structure, it is actually a method defined in `Kernel`. The block which follows introduces a new scope for local variables.

Ruby doesnt have a post-test loop

Ruby does not have a `do { ... } while` construct, so how can I implement loops that test the condition at the end?

Clemens Hintze says: "You can use a combination of Rubys `begin ... end` and the `while` or `until` statement modifiers to achieve the same effect:

```
i = 0
begin
  puts "i = #{i}"
  i += 1
end until i > 4
```

Why cant I pass a hash literal to a method: `p {}`?

The `{}` is parsed as a block, not a `Hash` constructor. You can force the `{}` to be treated as an expression by making the fact that it's a parameter explicit: `p({})`.

I cant get `def pos=(val)` to work!

I have the following code, but I cannot use the method `pos = 1`.

```
def pos=(val)
  @pos = val
  puts @pos
end
```

Methods with `=` appended must be called with an explicit receiver (without the receiver, you are just assigning to a local variable). Invoke it as `self.pos = 1`.

What is the difference between `\1` and `\\1`?

They have the same meaning. In a single quoted string, only `\'` and `\\` are transformed and other combinations remain unchanged.

However, in a double quoted string, `"\1"` is the byte `\001` (an octal bit pattern), while `"\\1"` is the two character string containing a backslash and the character `"1"`.

What is the difference between `..` and `...`?

`..` includes the right hand side in the range, while `...` does not:

```
(5..8).to_a   # => [5, 6, 7, 8]
(5...8).to_a  # => [5, 6, 7]
```

What is the difference between `or` and `||`?

`p(nil || "Hello")` prints `"Hello"`, while `p(nil or "Hello")` gives a parse error. Why?

`or` has a very low precedence; `p( (nil or "Hello") )` will work.

The precedence of `or` is for instance also lower than that of `=`, whereas `||` has a higher precedence:

```
foo = nil || "Hello"  # parsed as: foo = (nil || "Hello")
foo  # => "Hello"

# but perhaps surprisingly:

foo = nil or "Hello"  # parsed as: (foo = nil) or "Hello"
foo  # => nil
```

`or` (and similarly `and`) is best used, not for combining boolean expressions, but for control flow, like in:

```
do_something  or raise "some error!"
```

where `do_something` returns `false` or `nil` when an error occurs.

Does Ruby have function pointers?

A `Proc` object generated by `Proc.new`, `proc`, or `lambda` can be referenced from a variable, so that variable could be said to be a function pointer. You can also get references to methods within a particular object instance using `object.method`.

What is the difference between load and require?

`load` will load and execute a Ruby program (`*.rb`).

`require` loads Ruby programs as well, but will also load *binary Ruby extension modules* (shared libraries or DLLs). In addition, `require` ensures that a feature is never loaded more than once.

Does Ruby have exception handling?

Ruby supports a flexible exception handling scheme:

```
begin
  statements which may raise exceptions
rescue [exception class names]
  statements when an exception occurred
rescue [exception class names]
  statements when an exception occurred
ensure
  statements that will always run
end
```

If an exception occurs in the `begin` clause, the `rescue` clause with the matching exception name is executed. The `ensure` clause is executed whether an exception occurred or not. `rescue` and `ensure` clauses may be omitted.

If no exception class is designated for a `rescue` clause, `StandardError` exception is implied, and exceptions which are in a `is_a?` relation to `StandardError` are captured.

This expression returns the value of the `begin` clause.

The latest exception is accessed by the global variable `$!` (and so its type can be determined using `$!.type`).

2.3.2.3 FAQ Methods

How does Ruby choose which method to invoke?

Are `+`, `-`, `*`, `...` operators?

Where are `++` and `--` ?

What is a singleton method?

All these objects are fine, but does Ruby have any simple functions?

So where do all these function-like methods come from?

Can I access an objects instance variables?

Whats the difference between `private` and `protected`?

How can I change the visibility of a method?

Can an identifier beginning with a capital letter be a method name?

Calling `super` gives an `ArgumentError`.

How can I call the method of the same name two levels up?

How can I invoke an original built-in method after redefining it?

What is a destructive method?

Why can destructive methods be dangerous?

Can I return multiple values from a method?

2.3.2.4 FAQ Classes and Modules

Can a class definition be repeated?

Are there class variables?

What is a class instance variable?

What is the difference between class variables and class instance variables?

Does Ruby have class methods?

What is a singleton class?

What is a module function?

What is the difference between a class and a module?

Can you subclass modules?

Give me an example of a mixin

Why are there two ways of defining class methods?

What is the difference between include and extend?

What does `self` mean?

2.3.2.5 FAQ Built-In Libraries

What does `instance_methods(false)` return?

How do random number seeds work?

I read a file and changed it, but the file on disk has not changed.

How can I process a file and update its contents?

I wrote a file, copied it, but the end of the copy seems to be lost.

How can I get the line number in the current input file?

How can I use `less` to display my programs output?

What happens to a `File` object which is no longer referenced?

I feel uneasy if I dont close a file.

How can I sort files by their modification time?

How can I count the frequency of words in a file?

How can I sort strings in alphabetical order?

How can I expand tabs to spaces?

How can I escape a backslash in a regular expression?

What is the difference between `sub` and `sub!`?

Where does `\Z` match?

What is the difference between `thread` and `fork`?

How can I use `Marshal`?

How can I use `trap`?

2.3.2.6 FAQ Extension Library

How can I use Ruby interactively?

Is there a debugger for Ruby?

How can I use a library written in C from Ruby?

Can I use `Tcl/Tk` in Ruby?

`Tk` won't work. Why?

Can I use `gtk+` or `xforms` interfaces in Ruby?

How can I do date arithmetic?

2.3.2.7 FAQ Other Features

What does a ? b : c mean?

How can I count the number of lines in a file?

What do `MatchData#begin` and `MatchData#end` return?

How can I sum the elements in an array?

How can I use continuations?

2.3.3 Ruby Koans

Ruby Koans

The Koans walk you along the path to enlightenment in order to learn Ruby. The goal is to learn the Ruby language, syntax, structure, and some common functions and libraries. We also teach you culture.

2.3.4 Whys (Poignant) Guide to Ruby

Why's Guide to Ruby

An unconventional but interesting book that will teach you Ruby through stories, wit, and comics. Originally created by *why the lucky stiff*, this guide remains a classic for Ruby learners.

2.3.5 Ruby in Twenty Minutes

Ruby in Twenty Minutes

A nice tutorial covering the basics of Ruby. From start to finish it shouldnt take you more than twenty minutes. It makes the assumption that you already have Ruby installed. (If you do not have Ruby on your computer install it before you get started.)

2.3.5.1 Interactive Ruby

Ruby comes with a program that will show the results of any Ruby statements you feed it. Playing with Ruby code in interactive sessions like this is a terrific way to learn the language.

Open up `IRB` (which stands for Interactive Ruby).

```
? irb
⊣ irb(main):001:0>

irb(main):001:0> "Hello World"
⇒ "Hello World"
⊣ irb(main):002:0>
```

The second line is just IRBs way of telling us the result of the last expression it evaluated. To print:

```
irb(main):002:0> puts "Hello World"
⊣ Hello World
⇒ nil
⊣ irb(main):003:0>
```

`puts` is the basic command to print something out in Ruby. But then whats the '`=> nil`' bit? Thats the result of the expression. `puts` always returns `nil`, which is Rubys absolutely-positively-nothing value.

2.3.5.2 Defining Methods

Define a method:

```
irb(main):010:0> def hi
irb(main):011:1> puts "Hello World!"
irb(main):012:1> end
=> :hi
```

The code '`def hi`' starts the definition of the method. The next line is the body of the method. Finally, the last line end tells Ruby were done defining the method. Rubys response ⊣ `=> :hi` tells us that it knows we're done defining the method.

Try running that method a few times:

```
irb(main):013:0> hi
Hello World!
=> nil
irb(main):014:0> hi()
Hello World!
=> nil
```

If the method doesn't take parameters that's all you need. You can add empty parentheses if youd like, but theyre not needed.

Define Method with a Parameter

What if we want to say hello to one person, and not the whole world? Just redefine `hi` to take a name as a parameter.

```
irb(main):015:0> def hi(name)
irb(main):016:1> puts "Hello #{name}!"
irb(main):017:1> end
=> :hi
irb(main):018:0> hi("Matz")
Hello Matz!
=> nil
```

What's the `#{name}` bit? That's Ruby's way of inserting something into a string. The bit between the braces is turned into a string (if it isnt one already) and then substituted into the outer string at that point. You can also use this to make sure that someone's name is properly capitalized:

```
irb(main):019:0> def hi(name = "World")
irb(main):020:1> puts "Hello #{name.capitalize}!"
irb(main):021:1> end
=> :hi
irb(main):022:0> hi "chris"
Hello Chris!
=> nil
irb(main):023:0> hi
```

```
Hello World!
=> nil
```

A couple of other tricks to spot here. One is that we're calling the method without parentheses again. If it's obvious what youre doing, the parentheses are optional. The other trick is the default parameter World. What this is saying is "If the name isn't supplied, use the default name of "World"".

Create a Class

What if we want a real greeter around, one that remembers your name and welcomes you and treats you always with respect. You might want to use an object for that. Lets create a Greeter class.

```
irb(main):024:0> class Greeter
irb(main):025:1>   def initialize(name = "World")
irb(main):026:2>     @name = name
irb(main):027:2>   end
irb(main):028:1>   def say_hi
irb(main):029:2>     puts "Hi #{@name}!"
irb(main):030:2>   end
irb(main):031:1>   def say_bye
irb(main):032:2>     puts "Bye #{@name}, come back soon."
irb(main):033:2>   end
irb(main):034:1> end
=> :say_bye
```

The new keyword here is `class`. This defines a new class called `Greeter` and a bunch of methods for that class. Also notice `@name`. This is an instance variable, and is available to all the methods of the class. As you can see its used by `say_hi` and `say_bye`.

Create an Object

Now lets create a greeter object and use it:

```
irb(main):035:0> greeter = Greeter.new("Pat")
=> #<Greeter:0x16cac @name="Pat">
irb(main):036:0> greeter.say_hi
Hi Pat!
=> nil
irb(main):037:0> greeter.say_bye
Bye Pat, come back soon.
=> nil
```

Instance Variables

Instance variables are hidden away inside the object. Theyre not terribly hidden, you see them whenever you inspect the object, and there are other ways of accessing them, but Ruby uses the good object-oriented approach of keeping data sort-of hidden away.

So what methods do exist for Greeter objects?

'`Object#instance_methods`'

```
irb(main):039:0> Greeter.instance_methods
```

```
=> [:say_hi, :say_bye, :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, :define_singleton_method, :to_enum,
    :enum_for, :<=>, :===, :=~, :!~, :eql?, :respond_to?,
    :freeze, :inspect, :display, :send, :object_id, :to_s,
    :method, :public_method, :singleton_method, :nil?, :hash,
    :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__]
```

We only defined two methods. Whats going on here? Well this is all of the methods for Greeter objects, a complete list, including ones defined by ancestor classes. If we want to just list methods defined for Greeter we can tell it to not include ancestors by passing it the parameter false, meaning we dont want methods defined by ancestors.

'`Object#instance_methods(false)`'

```
irb(main):040:0> Greeter.instance_methods(false)
=> [:say_hi, :say_bye]
```

So lets see which methods our greeter object responds to:

'`Object#respond_to?`'

```
irb(main):041:0> greeter.respond_to?("name")
=> false
irb(main):042:0> greeter.respond_to?("say_hi")
=> true
irb(main):043:0> greeter.respond_to?("to_s")
=> true
```

So, it knows `say_hi`, and `to_s` (meaning convert something to a string, a method that's defined by default for every object), but it doesn't know `name`.

2.3.5.3 Altering Classes

But what if you want to be able to view or change the name? Ruby provides an easy way of providing access to an object's variables.

'`attr_accessor :name`'

```
irb(main):044:0> class Greeter
irb(main):045:1>   attr_accessor :name
irb(main):046:1> end
=> nil
```

In Ruby, you can open a class up again and modify it. The changes will be present in any new objects you create and even available in existing objects of that class. So, lets create a new object and play with its `@name` property.

```
irb(main):047:0> greeter = Greeter.new("Andy")
=> #<Greeter:0x3c9b0 @name="Andy">
```

```
irb(main):048:0> greeter.respond_to?("name")
=> true
irb(main):049:0> greeter.respond_to?("name=")
=> true
irb(main):050:0> greeter.say_hi
Hi Andy!
=> nil
irb(main):051:0> greeter.name="Betty"
=> "Betty"
irb(main):052:0> greeter
=> #<Greeter:0x3c9b0 @name="Betty">
irb(main):053:0> greeter.name
=> "Betty"
irb(main):054:0> greeter.say_hi
Hi Betty!
=> nil
```

Using `attr_accessor` defined two new methods for us, `name` to get the value, and `name=` to set it.

2.3.5.4 Large Class Definition

What if we had some kind of MegaGreeter that could either greet the world, one person, or a whole list of people? Lets write this one in a file instead of directly in the interactive Ruby interpreter IRB.

{ri20min.rb} ≡

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

class MegaGreeter
  attr_accessor :names

  <MegaGreeter—Initialize Method>
  <MegaGreeter—say_hi Method>
  <MegaGreeter—say_bye Method>
end

if __FILE__ == $0
  <MegaGreeter—Main Script>
end
```

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*MegaGreeter—Initialize Method*>	See "Large Class Definition", page 22.
<*MegaGreeter—Main Script*>	See "Large Class Definition", page 23.
<*MegaGreeter—say_bye Method*>	See "Large Class Definition", page 23.
<*MegaGreeter—say_hi Method*>	See "Large Class Definition", page 22.

Initialize Method

<*MegaGreeter—Initialize Method*> ≡

```
# Create the object
def initialize(names = "World")
  @names = names
end
```

This chunk is called by {`ri20min.rb`}; see its first definition at "Large Class Definition", page 21.

`say_hi` Method

The `say_hi` method has become a bit more complicated. It now looks at the `@names` instance variable to make decisions. If it's `nil`, it just prints out three dots. No point greeting nobody, right?

If the `@names` object responds to each, it is something that you can iterate over, so iterate over it and greet each person in turn. Finally, if `@names` is anything else, just let it get turned into a string automatically and do the default greeting.

<*MegaGreeter—say_hi Method*> ≡

```
# Say hi to everybody
def say_hi
  if @names.nil?
    puts "..."
  elsif @names.respond_to?("each")
    # @names is a list of some kind, iterate!
    @names.each do |name|
      puts "Hello #{name}!"
    end
  else
    puts "Hello #{@names}!"
  end
end
```

This chunk is called by {`ri20min.rb`}; see its first definition at "Large Class Definition", page 21.

The Iterator

Lets look at that iterator in more depth:

```
@names.each do |name|
  puts "Hello #{name}!"
end
```

`each` is a method that accepts a block of code then runs that block of code for every element in a list, and the bit between `do` and `end` is just such a block. A *block* is like an anonymous function or lambda. The variable between pipe characters is the parameter for this block.

What happens here is that for every entry in a list, `name` is bound to that list element, and then the expression puts '`"Hello #{name}!"`' is run with that name.

Internally, the `each` method will essentially call `yield` ‘`"Albert"`’, then `yield` ‘`"Brenda"`’ and then `yield` ‘`"Charles"`’, and so on.

The Real Power of Blocks

The real power of blocks is when dealing with things that are more complicated than lists. Beyond handling simple housekeeping details within the method, you can also handle setup, teardown, and errorsall hidden away from the cares of the user.

`say_bye` Method

The `say_bye` method doesn't use `each`; instead it checks to see if `@names` *responds* to the `join` method, and if so, uses it. Otherwise, it just prints out the variable as a string.

Duck Typing

This method of not caring about the actual type of a variable, just relying on what methods it supports is known as *Duck Typing*, as in “if it walks like a duck and quacks like a duck. . .”. The benefit of this is that it doesn't unnecessarily restrict the types of variables that are supported. If someone comes up with a new kind of list class, as long as it implements the `join` method with the same semantics as other lists, everything will work as planned.

<*MegaGreeter—say_bye Method*> ≡

```
# Say bye to everybody
def say_bye
  if @names.nil?
    puts "..."
  elsif @names.respond_to?("join")
    # Join the list elements with commas
    puts "Goodbye #{@names.join(", ")}.  Come back soon!"
  else
    puts "Goodbye #{@names}.  Come back soon!"
  end
end
```

This chunk is called by {`ri20min.rb`}; see its first definition at “Large Class Definition”, page 21.

MegaGreeter Main Script

Theres one final trick to notice, and thats the line:

```
if __FILE__ == $0
```

__FILE__ is the magic variable that contains the name of the current file. $0 is the name of the file used to start the program. This check says “If this is the main file being used. . .” This allows a file to be used as a library, and not to execute code in that context, but if the file is being used as an executable, then execute that code.

<*MegaGreeter—Main Script*> ≡

```
mg = MegaGreeter.new
mg.say_hi
mg.say_bye
```

```
# Change name to be "Zeke"
mg.names = "Zeke"
mg.say_hi
mg.say_bye

# Change the name to an array of names
mg.names = ["Albert", "Brenda", "Charles",
            "Dave", "Engelbert"]
mg.say_hi
mg.say_bye

# Change to nil
mg.names = nil
mg.say_hi
mg.say_bye
```

This chunk is called by {`ri20min.rb`}; see its first definition at "Large Class Definition", page 21.

2.3.5.5 Run MegaGreeter

Run the program `ri20min.rb` as '`ruby ri20min.rb`'. The output should be:

```
Hello World!
Goodbye World.  Come back soon!
Hello Zeke!
Goodbye Zeke.  Come back soon!
Hello Albert!
Hello Brenda!
Hello Charles!
Hello Dave!
Hello Engelbert!
Goodbye Albert, Brenda, Charles, Dave, Engelbert.  Come back soon!
...
...
```

2.3.6 Ruby from Other Languages

Ruby from Other Languages

2.3.7 Learning Ruby

Learning Ruby

A thorough collection of Ruby study notes for those who are new to the language and in search of a solid introduction to Rubys concepts and constructs.

2.3.8 Ruby Essentials

Ruby Essentials

2.3.9 Learn to Program

Learn to Program

A wonderful little tutorial by Chris Pine for programming newbies. If you dont know how to program, start here.

Learn Ruby the Hard Way

2.4 Manuals

2.5 Reference Documentation

2.6 Editors and IDEs

2.7 Further Reading

Appendix A The Makefile

{`Makefile`} ≡

<*Makefile—Variable Definitions*>
<*Makefile—Default Rule*>
<*Makefile—TWJR Rules*>
<*Makefile—Clean Rules*>

The following table lists called chunk definition points.

Chunk name	First definition point
<*Makefile—Clean Rules*>	See "Clean Rules", page 27.
<*Makefile—Default Rule*>	See "Default Rule", page 26.
<*Makefile—TWJR Rules*>	See "TWJR Rules", page 26.
<*Makefile—Variable Definitions*>	See "Makefile Variable Definitions", page 26.

A.1 Makefile Variable Definitions

<*Makefile—Variable Definitions*> ≡

```
FILE := Ruby2_5
SHELL := /bin/bash
```

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 26.

A.2 Default Rule

The `default` rule is to create a PDF document and all HTML files. This assumes that the TEXI file has been generated and updated by hand first. Therefore, the target `TWJR` will run both `jrtangle` and `jrweave`, while the target `WEAVE` or alternatively `TEXI` will run just `jrweave` on the `.twjr` file. Thereafter, you can update the `.texi` file and run the `default`.

<*Makefile—Default Rule*> ≡

```
.PHONY : default TWJR TANGLE WEAVE TEXI PDF HTML
.PHONY : twjr tangle weave texi pdf html
default : PDF HTML
```

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 26.

A.3 TWJR Rules

<*Makefile—TWJR Rules*> ≡

```
TWJR : twjr
twjr : tangle weave

TANGLE : tangle
tangle : $(FILE).twjr
    jrtangle $(FILE).twjr

WEAVE : weave
```

```
weave : TEXI
TEXI  : texi
texi  : $(FILE).texi

$(FILE).texi : $(FILE).twjr
    jrweave $(FILE).twjr > $(FILE).texi

PDF : pdf
pdf : $(FILE).pdf
$(FILE).pdf : $(FILE).texi
    pdftexi2dvi $(FILE).texi
    make distclean

HTML : html
html : $(FILE)/
$(FILE)/ : $(FILE).texi
    makeinfo --html $(FILE).texi
```

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 26.

A.4 Clean Rules

<*Makefile—Clean Rules*> ≡

```
.PHONY : clean distclean veryclean worldclean
clean :
    rm -f *~ \#*\#

distclean : clean
    rm -f *.{aux,log,toc,cp,cps}

veryclean : clean
    for file in *; do [[ $$file =~ $(FILE)|Makefile ]] && : || rm -vrf $$file ; done;

worldclean : veryclean
    rm -fr $(FILE).{texi,info,pdf} $(FILE)/
```

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 26.

Appendix B Code Chunk Summaries

This appendix presents alphabetical lists of all the file definitions, the code chunk definitions, and the code chunk references.

B.1 Source File Definitions

{`Makefile`}
: This chunk is defined in "The Makefile", page 26.

{`ri2Omin.rb`}
: This chunk is defined in "Large Class Definition", page 21.

B.2 Code Chunk Definitions

<*Makefile—Clean Rules*>
: This chunk is defined in "Clean Rules", page 27.

<*Makefile—Default Rule*>
: This chunk is defined in "Default Rule", page 26.

<*Makefile—TWJR Rules*>
: This chunk is defined in "TWJR Rules", page 26.

<*Makefile—Variable Definitions*>
: This chunk is defined in "Makefile Variable Definitions", page 26.

<*MegaGreeter—Initialize Method*>
: This chunk is defined in "Large Class Definition", page 22.

<*MegaGreeter—Main Script*>
: This chunk is defined in "Large Class Definition", page 23.

<*MegaGreeter—say_bye Method*>
: This chunk is defined in "Large Class Definition", page 23.

<*MegaGreeter—say_hi Method*>
: This chunk is defined in "Large Class Definition", page 22.

B.3 Code Chunk References

<*Makefile—Clean Rules*>
: This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 26.

<*Makefile—Default Rule*>
: This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 26.

<*Makefile—TWJR Rules*>
: This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 26.

<*Makefile—Variable Definitions*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 26.

<*MegaGreeter—Initialize Method*>
This chunk is called by {`ri20min.rb`}; see its first definition at "Large Class Definition", page 21.

<*MegaGreeter—Main Script*>
This chunk is called by {`ri20min.rb`}; see its first definition at "Large Class Definition", page 21.

<*MegaGreeter—say_bye Method*>
This chunk is called by {`ri20min.rb`}; see its first definition at "Large Class Definition", page 21.

<*MegaGreeter—say_hi Method*>
This chunk is called by {`ri20min.rb`}; see its first definition at "Large Class Definition", page 21.

Bibliography

Index

E

F

G

I

J

K

L

M

N

O

P

R

S

T

U

V

X

Y