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 0.2c 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

This document contains two major sections. The first attempts to be a rapid-fire summary of what you can expect to see when going from language X to Ruby. The second section tackles the major language features and how they might compare to what youre already familiar with.

2.3.6.1 To Ruby From C and C++

Everything Is Differerent

It's difficult to write a bulleted list describing how your code will be different in Ruby from C or C++ because it's quite a large difference. One reason is that the Ruby runtime does so much for you. Ruby seems about as far as you can get from C's "no hidden mechanism"

principle—the whole point of Ruby is to make the human's job easier at the expense of making the runtime shoulder more of the work.

Ruby is Quicker to Code But Slower to Execute

That said, for one thing, you can expect your Ruby code to execute much more slowly than "equivalent" C or C++ code. At the same time, your head will spin at how rapidly you can get a Ruby program up and running, as well as at how few lines of code it will take to write it. Ruby is much much simpler than C++.

Dynamically Typed

Ruby is dynamically typed, rather than statically typed—the runtime does as much as possible at run-time. For example, you don't need to know what modules your Ruby program will "link to" (that is, load and use) or what methods it will call ahead of time.

Extension Modules

Happily, it turns out that Ruby and C have a healthy symbiotic relationship. Ruby supports so-called *extension modules*. These are modules that you can use from your Ruby programs (and which, from the outside, will look and act just like any other Ruby module), but which are written in C. In this way, you can compartmentalize the performance-critical parts of your Ruby software, and smelt those down to pure C.

And, of course, Ruby itself is written in C.

Similarities With C

- You may program procedurally if you like (but it will still be object-oriented behind the scenes).
- Most of the operators are the same (including the compound assignment and also bitwise operators). Though, Ruby doesnt have `++` or `--`.
- Ruby has *__FILE__* and *__LINE__*.
- You can also have constants, though theres no special const keyword. `Const`-ness is enforced by a naming convention instead — names starting with a capital letter are for constants.
- Strings go in double-quotes and are mutable
- Just like man pages, you can read most docs in your terminal window — though using the `ri` command.
- Youve got the same sort of command-line debugger available.

Similarities with C++

- Youve got mostly the same operators (even `::`). `<<` is often used for appending elements to a list. One note though: with Ruby you never use `->` — it's always just `.`.
- `public`, `private`, and `protected` do similar jobs.
- Inheritance syntax is still only one character, but it's `<` instead of `:`.
- You may put your code into "modules", similar to how `namespace` in C++ is used.
- Exceptions work in a similar manner, though the keyword names have been changed to protect the innocent.

Differences From C

- You dont need to compile your code. You just run it directly.
- Objects are strongly typed (and variable names themselves have no type at all).
- Theres no macros or preprocessor; no casts; no pointers (nor pointer arithmetic); no typedefs, sizeof, or enums.
- There are no header files. You just define your functions (usually referred to as "methods") and classes in the main source code files.
- Theres no `#define`. Just use constants instead.
- All variables live on the heap. Further, you dont need to free them yourself — the garbage collector takes care of that.
- Arguments to methods (i.e. functions) are passed by value, where the values are always object references.
- It's '`require 'foo'`' instead of '`#include <foo>`' or '`#include "foo"`'.
- You cannot drop down to assembly.
- Theres no semicolons ending lines.
- You go without parentheses for `if` and `while` condition expressions.
- Parentheses for method (i.e. function) calls are often optional.
- You dont usually use braces — just end multi-line constructs (like `while` loops) with an `end` keyword.
- The `do` keyword is for so-called *blocks*. Theres no "do statement" like in C.
- The term *block* means something different. It's for a block of code that you associate with a method call so the method body can call out to the block while it executes.
- There are no variable declarations. You just assign to new names on-the-fly when you need them.
- When tested for truth, only `false` and `nil` evaluate to a `false` value. Everything else is true (including `0`, `0.0`, and `"0"`).
- There is no `char` — they are just 1-letter strings.
- Strings dont end with a null byte.
- Array literals go in brackets instead of braces.
- Arrays just automatically get bigger when you stuff more elements into them.
- If you add two arrays, you get back a new and bigger array (of course, allocated on the heap) instead of doing pointer arithmetic.
- More often than not, everything is an expression (that is, things like `while` statements actually evaluate to an `rvalue`).

Differences from C++

- Theres no explicit references. That is, in Ruby, every variable is just an automatically dereferenced name for some object.
- Objects are strongly but *dynamically* typed. The runtime discovers *at runtime* if that method call actually works.
- The *constructor* is called `initialize` instead of the class name.

- All methods are always virtual.
- "Class" (`static`) variable names always begin with `@@` (as in `@@total_widgets`).
- You dont directly access member variables — all access to public member variables (known in Ruby as *attributes*) is via methods.
- It's `self` instead of `this`.
- Some methods end in a `?` or a `!`. It's actually part of the method name.
- There's no multiple inheritance per se. Though Ruby has *mixins* (i.e. you can "inherit" all instance methods of a module).
- There are some enforced case-conventions (ex. class names start with a capital letter, variables start with a lowercase letter).
- Parentheses for method calls are usually optional.
- You can re-open a class anytime and add more methods.
- Theres no need of C++ templates (since you can assign any kind of object to a given variable, and types get figured out at runtime anyway). No casting either.
- Iteration is done a bit differently. In Ruby, you don't use a separate iterator object (like `vector<T>::const_iterator iter`). Instead you use an iterator method of the container object (like `each`) that takes a block of code to which it passes successive elements.
- Theres only two container types: `Array` and `Hash`.
- Theres no type conversions. With Ruby though, youll probably find that they arent necessary.
- Multithreading is built-in, but as of Ruby 1.8 they are *green threads* (implemented only within the interpreter) as opposed to native threads.
- A unit testing lib comes standard with Ruby.

2.3.6.2 To Ruby From Java

Ruby is Less Verbose

Java is mature. It's tested. And it's fast (contrary to what the anti-Java crowd may still claim). It's also quite verbose. Going from Java to Ruby, expect your code size to shrink down considerably. You can also expect it to take less time to knock together quick prototypes.

Similarities with Java

- Memory is managed for you via a garbage collector.
- Objects are strongly typed.
- There are `public`, `private`, and `protected` methods.
- There are embedded doc tools (Ruby's is called `RDoc`). The docs generated by `rdoc` look very similar to those generated by `javadoc`.

Differences From Java

- You dont need to compile your code. You just run it directly.

- There are several different popular third-party GUI toolkits. Ruby users can try `WxRuby`, `FXRuby`, `Ruby-GNOME2`, `Qt`, or the bundled-in `Ruby Tk` for example.
- You use the `end` keyword after defining things like classes, instead of having to put braces around blocks of code.
- You have `require` instead of `import`.
- All member variables are private. From the outside, you access everything via methods.

- Parentheses in method calls are usually optional and often omitted.
- Everything is an object, including numbers like `2` and `3.14159`.
- Theres no static type checking.
- Variable names are just labels. They dont have a type associated with them.
- There are no type declarations. You just assign to new variable names as-needed and they just "spring up" (i.e. '`a = [1,2,3]`' rather than '`int[] a = {1,2,3};`').
- Theres no casting. Just call the methods. Your unit tests should tell you before you even run the code if youre going to see an exception.
- It's '`foo = Foo.new("hi")`' instead of '`Foo foo = new Foo("hi")`'.
- The constructor is always named `initialize` instead of the name of the class.
- You have "mixins" instead of interfaces.
- YAML tends to be favored over XML.
- It's `nil` instead of `null`.
- `==` and `equals()` are handled differently in Ruby. Use `==` when you want to test "equivalence" in Ruby (`equals()` in Java). Use `equal?()` when you want to know if two objects are "the same" (`==` in Java).

2.3.6.3 To Ruby From Perl

Perl is awesome. Perl's docs are awesome. The Perl community is — awesome. However, the language is fairly large and arguably complex. For those Perlers who long for a simpler time, a more orthogonal language, and elegant OO features built-in from the beginning, Ruby may be for you.

Similarities with Perl

- Youve got a package management system, somewhat like CPAN (though its called RubyGems).
- Regexes are built right in.
- There's a fairly large number of commonly-used built-ins.
- Parentheses are often optional.
- Strings work basically the same.
- Theres a general delimited string and regex quoting syntax similar to Perls. It looks like `%q{this}` (single-quoted), or `%Q{this}` (double-quoted), and `%w{this for a single-quoted list of words}`. You `%Q|can|` `%Q(use)` `%Q^other^` delimiters if you like.

- Youve got double-quotish variable interpolation, though it '`"looks #{like} this"`' (and you can put any Ruby code you like inside that `#{}`).
- Shell command expansion uses `` `backticks` ``.
- Youve got embedded doc tools (Rubys is called `rdoc`).

Differences From Perl

- You dont have the context-dependent rules like with Perl.
- A variable isn't the same as the object to which it refers. Instead, it's always just a reference to an object.
- Although `$` and `@` are used as the first character in variable names sometimes, rather than indicating type, they indicate scope (`$` for globals, `@` for object instance, and `@@` for class attributes).
- Array literals go in brackets instead of parentheses.
- Composing lists of other lists does not flatten them into one big list. Instead you get an array of arrays.
- It's `def` instead of `sub`.
- There's no semicolons needed at the end of each line. Incidentally, you end things like function definitions, class definitions, and case statements with the `end` keyword.
- Objects are strongly typed. Youll be manually calling `foo.to_i`, `foo.to_s`, etc., if you need to convert between types.
- Theres no `eq`, `ne`, `lt`, `gt`, `ge`, nor `le`.
- Theres no diamond operator (`<>`). You usually use `IO.some_method` instead.
- The fat comma `=>` is only used for hash literals.
- Theres no `undef`. In Ruby you have `nil`. `nil` is an object (like anything else in Ruby). It's not the same as an undefined variable. It evaluates to `false` if you treat it like a boolean.
- When tested for truth, only `false` and `nil` evaluate to a `false` value. Everything else is `true` (including `0`, `0.0`, and `"0"`).

2.3.6.4 To Ruby From PHP

PHP is in widespread use for web applications, but if you want to use Ruby on Rails or just want a language thats more tailored for general use, Ruby is worth a look.

Similarities with PHP

- Ruby is dynamically typed, like in PHP, so you dont need to worry about having to declare variables.
- There are classes, and you can control access to them like in PHP 5 (public, protected and private).
- Some variables start with `$`, like in PHP (but not all).
- There's `eval`, too.
- You can use string interpolation. Instead of doing '`"$foo is a $bar"`', you can do '`"#{foo} is a #{bar}"`' — like in PHP, this doesnt apply for single-quoted strings.

- Theres heredocs.
- Ruby has exceptions, like PHP 5.
- Theres a fairly large standard library.
- Arrays and hashes work like expected, if you exchange `array()` for `{` and `}`: `array('a' => 'b')` becomes `{'a' => 'b'}`.
- `true` and `false` behave like in PHP, but `null` is called `nil`.

Differences From PHP

- Theres strong typing. Youll need to call `to_s`, `to_i` etc. to convert between strings, integers and so on, instead of relying on the language to do it.
- Strings, numbers, arrays, hashes, etc. are objects. Instead of calling `abs(-1)` its `-1.abs`.
- Parentheses are optional in method calls, except to clarify which parameters go to which method calls.
- The standard library and extensions are organized in modules and classes.
- Reflection is an inherent capability of objects; you dont need to use `Reflection` classes like in PHP 5.
- Variables are references.
- Theres no `abstract` classes or `interfaces`.
- Hashes and arrays are not interchangeable.
- Only `false` and `nil` are false: `0`, `array()` and `""` are all true in conditionals.
- Almost everything is a method call, even `raise` (`throw` in PHP).

2.3.6.5 To Ruby From Python

Python is another very nice general purpose programming language. Going from Python to Ruby, you'll find that there's a little bit more syntax to learn than with Python.

Similarities With Python

- Theres an interactive prompt (called `irb`).
- You can read docs on the command line (with the `ri` command instead of `pydoc`).
- There are no special line terminators (except the usual newline).
- String literals can span multiple lines like Python's triple-quoted strings.
- Brackets are for lists, and braces are for dicts (which, in Ruby, are called "hashes").
- Arrays work the same (adding them makes one long array, but composing them like this '`a3 = [ a1, a2 ]`' gives you an array of arrays).
- Objects are strongly and dynamically typed.
- Everything is an object, and variables are just references to objects.
- Although the keywords are a bit different, exceptions work about the same.
- Youve got embedded doc tools (Rubys is called `rdoc`).
- There is good support for functional programming with first-class functions, anonymous functions, and closures.

Differences From Python

- Strings are mutable.
- You can make constants (variables whose value you dont intend to change).
- There are some enforced case-conventions (ex. class names start with a capital letter, variables start with a lowercase letter).
- There's only one kind of list container (an `Array`), and it's mutable.
- Double-quoted strings allow escape sequences (like `\t`) and a special "expression substitution" syntax (which allows you to insert the results of Ruby expressions directly into other strings without having to '`"add " + "strings " + "together"`'). Single-quoted strings are like Python's `r"raw strings"`.
- There are no "new style" and "old style" classes. Just one kind. (Python 3+ doesnt have this issue, but it isnt fully backward compatible with Python 2.)
- You never directly access attributes. With Ruby, its all method calls.
- Parentheses for method calls are usually optional.
- There's `public`, `private`, and `protected` to enforce access, instead of Python's `_voluntary_` underscore `__convention__`.
- "mixins" are used instead of multiple inheritance.
- You can add or modify the methods of built-in classes. Both languages let you open up and modify classes at any point, but Python prevents modification of built-ins — Ruby does not.
- Youve got `true` and `false` instead of `True` and `False` (and `nil` instead of `None`).
- When tested for truth, only `false` and `nil` evaluate to a `false` value. Everything else is `true` (including `0`, `0.0`, `""`, and `[]`).
- It's `elsif` instead of `elif`.
- It's `require` instead of `import`. Otherwise though, usage is the same.
- The usual-style comments on the line(s) above things (instead of docstrings below them) are used for generating docs.
- There are a number of shortcuts that, although give you more to remember, you quickly learn. They tend to make Ruby fun and very productive.
- Theres no way to unset a variable once set (like Python's `del` statement). You can reset a variable to `nil`, allowing the old contents to be garbage collected, but the variable will remain in the symbol table as long as it is in scope.
- The `yield` keyword behaves differently. In Python it will return execution to the scope outside the function's invocation. External code is responsible for resuming the function. In Ruby `yield` will execute another function that has been passed as the final argument, then immediately resume.
- Python supports just one kind of anonymous functions, lambdas, while Ruby contains blocks, Procs, and lambdas.

2.3.7 Important Language Features

Here are some pointers and hints on major Ruby features you'll see while learning Ruby.

2.3.7.1 Pointers on Iteration

Two Ruby features that are a bit unlike what you may have seen before, and which take some getting used to, are "blocks" and iterators. Instead of looping over an index (like with C, C++, or pre-1.5 Java), or looping over a list (like Perl's `for (@a) {...}`, or Python's `for i in aList: ...`), with Ruby you'll very often instead see:

```
some_list.each do |this_item|
  # We're inside the block.
  # deal with this_item.
end
```

For more info on `each` and its friends

- `collect`,
- `find`,
- `inject`,
- `sort`,

etc., see `ri Enumerable` (and then `ri Enumerable#some_method`).

2.3.7.2 Everything has a value

Theres no difference between an expression and a statement. Everything has a value, even if that value is `nil`. This is possible:

```
x = 10
y = 11
z = if x < y
      true
    else
      false
    end
z # => true
```

2.3.7.3 Symbols are not lightweight Strings

Many Ruby newbies struggle with understanding what Symbols are, and what they can be used for.

Symbols can best be described as identities. A symbol is all about who it is, not what it is. Fire up `irb` and see the difference:

```
irb(main):001:0> :george.object_id == :george.object_id
=> true
irb(main):002:0> "george".object_id == "george".object_id
=> false
irb(main):003:0>
```

The `object_id` methods returns the identity of an Object. If two objects have the same `object_id`, they are the same (point to the same Object in memory).

As you can see, once you have used a Symbol once, any Symbol with the same characters references the same Object in memory. For any given two Symbols that represent the same characters, the `object_id`s match.

Now take a look at the String (`george`). The `object_id`s don't match. That means they're referencing two different objects in memory. Whenever you use a new String, Ruby allocates memory for it.

If you're in doubt whether to use a Symbol or a String, consider whats more important: the identity of an object (i.e. a Hash key), or the contents (in the example above, `george`).

2.3.7.4 Everything is an Object

```
''Everything is an object'' isn't just hyperbole.  Even classes and integers are
objects, and you can do the same things with them as with any other object:

# This is the same as
# class MyClass
#   attr_accessor :instance_var
# end
MyClass = Class.new do
  attr_accessor :instance_var
end
```

2.3.7.5 Variable Constants

Constants are not really constant. If you modify an already initialized constant, it will trigger a warning, but not halt your program. That isn't to say you should redefine constants, though.

2.3.7.6 Naming conventions

Ruby enforces some naming conventions. If an identifier starts with a capital letter, it is a constant. If it starts with a dollar sign (`$`), it is a global variable. If it starts with `@`, it is an instance variable. If it starts with `@@`, it is a class variable.

Method names, however, are allowed to start with capital letters. This can lead to confusion, as the example below shows:

```
Constant = 10
def Constant
  11
end
```

Now `Constant` is 10, but `Constant()` is 11.

2.3.7.7 Keyword arguments

Like in Python, since Ruby 2.0 methods can be defined using keyword arguments:

```
def deliver(from: "A", to: nil, via: "mail")
  "Sending from #{from} to #{to} via #{via}."
end

deliver(to: "B")
# => "Sending from A to B via mail."
deliver(via: "Pony Express", from: "B", to: "A")
# => "Sending from B to A via Pony Express."
```

2.3.7.8 The universal truth

In Ruby, everything except `nil` and `false` is considered true. In C, Python and many other languages, `0` and possibly other values, such as empty lists, are considered false. Take a look at the following Python code (the example applies to other languages, too):

```
# in Python
if 0:
  print("0 is true")
else:
  print("0 is false")
```

This will print '`0 is false`'. The equivalent Ruby:

```
# in Ruby
if 0
  puts "0 is true"
else
  puts "0 is false"
end
```

Prints '`0 is true`'.

2.3.7.9 Access modifiers are Methods

Access modifiers apply until the end of scope.

In the following Ruby code,

```
class MyClass
  private
  def a_method; true; end
  def another_method; false; end
end
```

You might expect `another_method` to be public. Not so. The `private` access modifier continues until the end of the scope, or until another access modifier pops up, whichever comes first. By default, methods are public:

```
class MyClass
  # Now a_method is public
  def a_method; true; end

  private

  # another_method is private
  def another_method; false; end
end
```

- `public`,
- `private` and
- `protected`

are really methods, so they can take parameters. If you pass a Symbol to one of them, that methods visibility is altered.

2.3.7.10 Method access

In Java, `public` means a method is accessible by anyone. `protected` means the class's instances, instances of descendant classes, and instances of classes in the same package can access it, but not anyone else; and `private` means nobody besides the class's instances can access the method.

Ruby differs slightly. `public` is, naturally, public. `private` means the method(s) are accessible only when they can be called without an explicit receiver. Only `self` is allowed to be the receiver of a private method call.

`protected` is the one to be on the lookout for. A `protected` method can be called from a class or descendant class instances, but also with another instance as its receiver. Here is an example (adapted from The Ruby Language FAQ):

```
class Test
  # public by default
  def identifier
    99
  end

  def ==(other)
    identifier == other.identifier
  end
end

t1 = Test.new  # => #<Test:0x34ab50>
t2 = Test.new  # => #<Test:0x342784>
t1 == t2       # => true

# now make ‘identifier’ protected; it still works
# because protected allows ‘other’ as receiver

class Test
  protected :identifier
end

t1 == t2  # => true

# now make ‘identifier’ private

class Test
  private :identifier
end

t1 == t2
# NoMethodError: private method ‘identifier’ called for #<Test:0x342784>
```

2.3.7.11 Classes are open

Ruby classes are open. You can open them up, add to them, and change them at any time. Even core classes, like `Fixnum` or even `Object`, the parent of all objects. Ruby on Rails defines a bunch of methods for dealing with time on `Fixnum`. Watch:

```
class Fixnum
  def hours
    self * 3600 # number of seconds in an hour
  end
  alias hour hours
end

# 14 hours from 00:00 January 1st
# (aka when you finally wake up ;)
Time.mktime(2006, 01, 01) + 14.hours # => Sun Jan 01 14:00:00
```

2.3.7.12 Funny method names

In Ruby, methods are allowed to end with question marks or exclamation marks. By convention, methods that answer questions end in question marks (e.g. `Array#empty?`, which returns `true` if the receiver is empty). Potentially "dangerous" methods by convention end with exclamation marks (e.g. methods that modify `self` or the arguments, `exit!`, etc.). Not all methods that change their arguments end with exclamation marks, though. `Array#replace` replaces the contents of an array with the contents of another array. It doesnt make much sense to have a method like that that doesn't modify self.

2.3.7.13 Singleton methods

Singleton methods are per-object methods. They are only available on the Object you defined it on.

```
class Car
  def inspect
    "Cheap car"
  end
end

porsche = Car.new
porsche.inspect # => Cheap car
def porsche.inspect
  "Expensive car"
end

porsche.inspect # => Expensive car

# Other objects are not affected
other_car = Car.new
other_car.inspect # => Cheap car
```

2.3.7.14 Missing methods

Ruby doesn't give up if it can't find a method that responds to a particular message. It calls the `method_missing` method with the name of the method it couldn't find and the arguments. By default, `method_missing` raises a `NameError` exception, but you can redefine it to better fit your application, and many libraries do. Here is an example:

```
# id is the name of the method called, the * syntax collects
# all the arguments in an array named 'arguments'
def method_missing(id, *arguments)
  puts "Method #{id} was called, but not found. It has " +
       "these arguments: #{arguments.join(", ")}"
end

__ :a, :b, 10
# => Method __ was called, but not found. It has these
# arguments: a, b, 10
```

The code above just prints the details of the call, but you are free to handle the message in any way that is appropriate.

2.3.7.15 Message passing, not function calls

A method call is really a "message" to another object:

```
# This
1 + 2
# Is the same as this ...
1.+(2)
# Which is the same as this:
1.send "+", 2
```

2.3.7.16 Blocks are Objects

Blocks (closures, really) are heavily used by the standard library. To call a block, you can either use `yield`, or make it a `Proc` by appending a special argument to the argument list, like so:

```
def block(&the_block)
  # Inside here, the_block is the block passed to the method
  the_block # return the block
end
adder = block { |a, b| a + b }
# adder is now a Proc object
adder.class # => Proc
```

You can create blocks outside of method calls, too, by calling `Proc.new` with a block or calling the `lambda` method.

Similarly, methods are also Objects in the making:

```
method(:puts).call "puts is an object!"
# => puts is an object!
```

2.3.7.17 Operators are syntactic sugar

Most operators in Ruby are just syntactic sugar (with some precedence rules) for method calls. You can, for example, override `Fixnum`s `+` method:

```
class Fixnum
  # You can, but please don't do this
  def +(other)
    self - other
  end
end
```

You dont need C++'s `operator+`, etc.

You can even have array-style access if you define the `[]` and `[]=` methods. To define the unary `+` and `-` (think '`+1`' and '`-2`'), you must define the `+@` and `-@` methods, respectively. The operators below are not syntactic sugar, though. They are not methods, and cannot be redefined:

```
=, .., ..., not, &&, and, ||, or, ::
```

In addition, '`+=, *=`' etc. are just abbreviations for '`var = var + other_var`', '`var = var * other_var`', etc. and therefore cannot be redefined.

2.3.8 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.

2.3.9 Ruby Essentials

Ruby Essentials

2.3.10 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

2.4.1 Ruby User's Guide

Translated from the original Japanese version written by Yukihiro Matsumoto (the creator of Ruby), this version, by Goto Kentaro and Mark Slagell, is a nice overview of many aspects of the Ruby language.

Ruby User's Guide

2.4.1.1 What Is Ruby?

Ruby is "an interpreted scripting language for quick and easy object-oriented programming" — what does this mean?

interpreted scripting language:

- ability to make operating system calls directly
- powerful string operations and regular expressions
- immediate feedback during development

quick and easy:

- variable declarations are unnecessary
- variables are not typed
- syntax is simple and consistent
- memory management is automatic

object oriented programming:

- everything is an object
- classes, methods, inheritance, etc.
- singleton methods
- "mixin" functionality by module
- iterators and closures

also:

- multiple precision integers
- convenient exception processing
- dynamic loading
- threading support

2.4.1.2 Simple Examples

Factorial in Ruby

Let's write a function to compute factorials. The mathematical definition of *n factorial* is:

```
n! = 1              (when n==0)
   = n * (n-1)!     (otherwise)
```

In ruby, this can be written as:

{`fact.rb`} ≡

```
# Program to find the factorial of a number
# Save this as fact.rb

def fact(n)
  if n == 0
    1
  else
    n * fact(n-1)
  end
end

puts fact(ARGV[0].to_i)
```

Command Line Arguments — In Array *ARGV*

`ARGV` is an array which contains the command line arguments, and `to_i` converts a character string to an integer.

The `end` Statement

You may notice the repeated occurrence of `end`. Ruby has been called "Algol-like" because of this. (Actually, the syntax of ruby more closely mimics that of a langage named Eiffel.)

Takeaway — `return` Statement Optional

You may also notice the lack of a `return` statement.

> [A `return` statement] is unneeded because **a ruby function returns the last thing that was evaluated in it**. Use of a `return` statement here is permissible but unnecessary.

Running `fact.rb`

Ruby can deal with any integer which is allowed by your machine's memory. So 400! can be calculated:

```
% ruby fact.rb 1
1
% ruby fact.rb 5
120

% ruby fact.rb 40
815915283247897734345611269596115894272000000000

% ruby fact.rb 400
64034522846623895262347970319503005850702583026002959458684
44594280239716918683143627847864746326467629435057503585681
08482981628835174352289619886468029979373416541508381624264
61942352307046244325015114448670890662773914918117331955996
44070954967134529047702032243491121079759328079510154537266
72516278778900093497637657103263503315339653498683868313393
52024373788157786791506311858702618270169819740062983025308
59129834616227230455833952075961150530223608681043329725519
48526744322324386699484224042325998055516106359423769613992
31917134063858996537970147827206606320217379472010321356624
61380907794230459736069956759583609615871512991382228657857
95493616176544804532220078258184008484364155912294542753848
03558374518022675900061399560145595206127211192918105032491
00800000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
```

The Input/Evaluation Loop

When you invoke ruby with no arguments, it reads commands from standard input and executes them after the end of input:

```
% ruby
```

```
puts "hello world"
puts "good-bye world"
^D
hello world
good-bye world
```

Ruby Evaluation Program — `eval.rb`

Ruby also comes with a program called `eval.rb` that allows you to enter ruby code from the keyboard in an interactive loop, showing you the results as you go. It will be used extensively through the rest of this guide. You should use this enhanced `eval.rb` that adds visual indenting assistance, warning reports, and color highlighting.

Here is a short `eval.rb` session:

```
% ruby eval.rb
ruby> puts "Hello, world."
Hello, world.
   nil
ruby> exit
```

'`hello world`' is produced by `puts`. The next line, in this case `nil`, reports on whatever was last evaluated;

No Distinction Between Statement and Expression

> Ruby does not distinguish between statements and expressions, so evaluating a piece of code basically means the same thing as executing it.

Here, `nil` indicates that `puts` does not return a meaningful value. Note that we can leave this interpreter loop by saying `exit`, although `C-D` still works too.

2.4.2 Programming Ruby

The Programmatic Programmer's Guide

Programming Ruby

What This Book Is

This book is a tutorial and reference for the Ruby programming language. Use Ruby, and you'll write better code, be more productive, and enjoy programming more.

What Ruby Is

Take a true object-oriented language, such as Smalltalk. Drop the unfamiliar syntax and move to more conventional, file-based source code. Now add in a good measure of the flexibility and convenience of languages such as Python and Perl.

You end up with Ruby.

Ruby is OO

OO aficionados will find much to like in Ruby: things such as pure object orientation (everything's an object), metaclasses, closures, iterators, and ubiquitous heterogeneous collections. Smalltalk users will feel right at home (and C++ and Java users will feel jealous).

Ruby is Perl and Python

At the same time, Perl and Python wizards will find many of their favorite features: full regular expression support, tight integration with the underlying operating system, convenient shortcuts, and dynamic evaluation.

Principle of Lease Surprise

Ruby follows the Principle of Least Surprise — things work the way you would expect them to, with very few special cases or exceptions.

Ruby is a "Transparent" Language

We call Ruby a "transparent" language. By that we mean that Ruby doesn't obscure the solutions you write behind lots of syntax and the need to churn out reams of support code just to get simple things done. With Ruby you write programs close to the problem domain. Rather than constantly mapping your ideas and designs down to the pedestrian level of most languages, with Ruby you'll find you can express them directly and express them elegantly. This means you code faster. It also means your programs stay readable and maintainable.

Ruby is a "Scripting" Language

What exactly is a scripting language? Frankly we don't know if it's a distinction worth making. In Ruby, you can access all the underlying operating system features. You can do the same stuff in Ruby that you can in Perl or Python, and you can do it more cleanly. But Ruby is fundamentally different. It is a true programming language, too, with strong theoretical roots and an elegant, lightweight syntax. You could hack together a mess of "scripts" with Ruby, but you probably won't. Instead, you'll be more inclined to engineer a solution, to produce a program that is easy to understand, simple to maintain, and a piece of cake to extend and reuse in the future.

Ruby is a General Purpose Programming Language

Although we have used Ruby for scripting jobs, most of the time we use it as a general-purpose programming language. We've used it to write GUI applications and middle-tier server processes, and we're using it to format large parts of this book. Others have used it for managing server machines and databases. Ruby is serving Web pages, interfacing to databases and generating dynamic content. People are writing artificial intelligence and machine learning programs in Ruby, and at least one person is using it to investigate natural evolution. Ruby's finding a home as a vehicle for exploratory mathematics. And people all over the world are using it as a way of gluing together all their different applications. It truly is a great language for producing solutions in a wide variety of problem domains.

Should I Use Ruby?

However, Ruby is probably more applicable than you might think. It is easy to extend, both from within the language and by linking in third-party libraries. It is portable across a number of platforms. It's relatively lightweight and consumes only modest system resources. And it's easy to learn; we've known people who've put Ruby code into production systems within a day of picking up drafts of this book. We've used Ruby to implement parts of an X11 window manager, a task that's normally considered severe C coding. Ruby excelled, and helped us write code in hours that would otherwise have taken days.

2.5 Reference Documentation

2.6 Editors and IDEs

2.7 Further Reading

Appendix A Utility Programs

A.1 eval.rb

eval.rb

{eval.rb} ≡

```
#!/usr/local/bin/ruby

##########################################################
#
# Ruby interactive input/eval loop
# Written by matz           (matz@netlab.co.jp)
# Modified by Mark Slagell  (slagell@ruby-lang.org)
#    with suggestions for improvement from Dave Thomas
#                           (Dave@Thomases.com)
#
##########################################################
#
# NOTE - this file has been renamed with a .txt extension to
# allow you to view or download it without the rubyist.net
# web server trying to run it as a CGI script.  You will
# probably want to rename it back to eval.rb.
#
##########################################################

module EvalWrapper

  <eval—EvalWrapper–Constants>

  <eval—EvalWrapper–Indentation Deltas>

  # On exit, restore normal screen colors.
  END {  print Norm,"\n" }

  ##################################################################
  # Execution starts here.
  ##################################################################

  indent=0
  while true   # Top of main loop.

    <eval—Main–Get Line>
    <eval—Main–Process Line>

  end          # Bottom of main loop
  print "\n"
```

```
end # module
```

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*eval—EvalWrapper–Constants*>	See “eval.rb Module Code”, page 45.
<*eval—EvalWrapper–Indentation Deltas*>	See “eval.rb Indentation Deltas Code”, page 45.
<*eval—Main–Get Line*>	See “eval.rb Main Get Line Code”, page 46.
<*eval—Main–Process Line*>	See “eval.rb Main Process Line Code”, page 46.

A.1.1 eval.rb Module Code

<*eval—EvalWrapper–Constants*> ≡

```
# Constants for ANSI screen interaction.  Adjust to your liking.

Norm   = "\033[0m"
PCol   = Norm           # Prompt color
Code   = "\033[1;32m"   # yellow
Eval   = "\033[0;36m"   # cyan
Prompt = PCol+"ruby> "+Norm
PrMore = PCol+"    | "+Norm
Ispace = "   "          # Adjust length of this for indentation.
Wipe   = "\033[A\033[K" # Move cursor up and erase line
```

This chunk is called by {eval.rb}; see its first definition at “eval.rb”, page 44.

A.1.2 eval.rb Indentation Deltas Code

<*eval—EvalWrapper–Indentation Deltas*> ≡

```
# Return a pair of indentation deltas. The first applies before
# the current line is printed, the second after.

def EvalWrapper.indentation( code )
  case code

  when /^\s*(class|module|def|if|case|while|for|begin)\b[^_]/
    [0,1]      # increase indentation because of keyword

  when /^\s*end\b[^_]/
    [-1,0]     # decrease because of end

  when /\{\s*(\|.*\|)?\s*$/
    [0,1]      # increase because of '{'

  when /^\s*\}/
    [-1,0]     # decrease because of '}'

  when /^\s*(rescue|ensure|elsif|else)\b[^_]/
    [-1,1]     # decrease for this line, then come back
```

```
  else
    [0,0]      # we see no reason to change anything

  end # case
end # def
```

This chunk is called by {`eval.rb`}; see its first definition at "`eval.rb`", page 44.

A.1.3 `eval.rb` Main Get Line Code

<*eval—Main–Get Line*> ≡

```
# Print prompt, move cursor to tentative indentation level, and get
# a line of input from the user.

if( indent == 0 )
  expr = ''; print Prompt  # (expecting a fresh expression)

else
  print PrMore             # (appending to previous lines)

end

print Ispace * indent,Code
line = gets
print Norm
```

This chunk is called by {`eval.rb`}; see its first definition at "`eval.rb`", page 44.

A.1.4 `eval.rb` Main Process Line Code

<*eval—Main–Process Line*> ≡

```
<eval—Main–Process Line-If Not Line>

<eval—Main–Process Line-Is Line>
```

This chunk is called by {`eval.rb`}; see its first definition at "`eval.rb`", page 44.

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*eval—Main–Process Line-If Not Line*>	See "`eval.rb` If Not Line Code", page 46.
<*eval—Main–Process Line-Is Line*>	See "`eval.rb` If Is Line Code", page 47.

A.1.4.1 `eval.rb` If Not Line Code

<*eval—Main–Process Line-If Not Line*> ≡

```
if not line
  # end of input (^D) - if there is no expression, exit, else
  # reset cursor to the beginning of this line.

  if expr == '' then break else print "\r" end
```

This chunk is called by <*eval—Main–Process Line*>; see its first definition at "`eval.rb` Main Process Line Code", page 46.

A.1.4.2 `eval.rb` If Is Line Code

<*eval—Main–Process Line-Is Line*> ≡

```
else

  # Append the input to whatever we had.
  expr << line

  <eval—Main–Process Line-Is Line_Indentation>

  <eval—Main–Process Line-Is Line_Worth Evaluating?>

end # if not line
```

This chunk is called by <*eval—Main–Process Line*>; see its first definition at "`eval.rb` Main Process Line Code", page 46.

The following table lists called chunk definition points.

Chunk name		First definition point
<*eval—Main–Process Line_Indentation*>	*Line-Is*	See "`eval.rb` If Is Line Code", page 47.
<*eval—Main–Process Line_Worth Evaluating?*>	*Line-Is*	See "`eval.rb` If Is Line Code", page 48.

Indentation

<*eval—Main–Process Line-Is Line_Indentation*> ≡

```
# Determine changes in indentation, reposition this line if
# necessary, and adjust indentation for the next prompt.

begin
  ind1,ind2 = indentation( line )
  if( ind1 != 0 )
    indent += ind1
    print Wipe,PrMore,(Ispace*indent),Code,line,Norm
  end
  indent += ind2

rescue      # On error, restart the main loop.
  print Eval,"ERR: Nesting violation\n",Norm
  indent = 0
  redo

end # begin
```

This chunk is called by <*eval—Main–Process Line-Is Line*>; see its first definition at "`eval.rb` If Is Line Code", page 47.

Something Worth Evaluating?

<*eval—Main–Process Line-Is Line_ Worth Evaluating?*> ≡

```
# Okay, do we have something worth evaulating?

if (indent == 0) && (expr.chop =~ /[^; \t\n\r\f]+/)

  begin
    result = eval(expr, TOPLEVEL_BINDING).inspect
    if $!  # no exception, but $! non-nil, means a warning
      print Eval,$!,Norm,"\n"
      $!=nil
    end
    print Eval,"   ",result,Norm,"\n"

  rescue ScriptError,StandardError
    $! = 'exception raised' if not $!
    print Eval,"ERR: ",$!,Norm,"\n"
  end

  break if not line

end # if
```

This chunk is called by <*eval—Main–Process Line-Is Line*>; see its first definition at "`eval.rb` If Is Line Code", page 47.

Appendix B 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 50.
<*Makefile—Default Rule*>	See "Default Rule", page 49.
<*Makefile—TWJR Rules*>	See "TWJR Rules", page 49.
<*Makefile—Variable Definitions*>	See "Makefile Variable Definitions", page 49.

B.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 49.

B.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 49.

B.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 49.

B.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 49.

Appendix C Code Chunk Summaries

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

C.1 Source File Definitions

{`Makefile`}
This chunk is defined in "The Makefile", page 49.

{`eval.rb`}
This chunk is defined in "`eval.rb`", page 44.

{`fact.rb`}
This chunk is defined in "Simple Examples", page 39.

{`ri20min.rb`}
This chunk is defined in "Large Class Definition", page 21.

C.2 Code Chunk Definitions

<*Makefile—Clean Rules*>
This chunk is defined in "Clean Rules", page 50.

<*Makefile—Default Rule*>
This chunk is defined in "Default Rule", page 49.

<*Makefile—TWJR Rules*>
This chunk is defined in "TWJR Rules", page 49.

<*Makefile—Variable Definitions*>
This chunk is defined in "Makefile Variable Definitions", page 49.

<*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.

<*eval—EvalWrapper–Constants*>
This chunk is defined in "`eval.rb` Module Code", page 45.

<*eval—EvalWrapper–Indentation Deltas*>
This chunk is defined in "`eval.rb` Indentation Deltas Code", page 45.

<*eval—Main–Get Line*>
This chunk is defined in "`eval.rb` Main Get Line Code", page 46.

<*eval—Main–Process Line*>
This chunk is defined in "`eval.rb` Main Process Line Code", page 46.

<*eval—Main–Process Line-If Not Line*>
This chunk is defined in "`eval.rb` If Not Line Code", page 46.

<*eval—Main–Process Line-Is Line*>
This chunk is defined in "`eval.rb` If Is Line Code", page 47.

<*eval—Main–Process Line-Is Line_Indentation*>
This chunk is defined in "`eval.rb` If Is Line Code", page 47.

<*eval—Main–Process Line-Is Line_ Worth Evaluating?*>
This chunk is defined in "`eval.rb` If Is Line Code", page 48.

C.3 Code Chunk References

<*Makefile—Clean Rules*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 49.

<*Makefile—Default Rule*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 49.

<*Makefile—TWJR Rules*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 49.

<*Makefile—Variable Definitions*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 49.

<*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.

<*eval—EvalWrapper–Constants*>
This chunk is called by {`eval.rb`}; see its first definition at "`eval.rb`", page 44.

<*eval—EvalWrapper–Indentation Deltas*>
This chunk is called by {`eval.rb`}; see its first definition at "`eval.rb`", page 44.

<*eval—Main–Get Line*>
This chunk is called by {`eval.rb`}; see its first definition at "`eval.rb`", page 44.

<*eval—Main–Process Line*>
This chunk is called by {`eval.rb`}; see its first definition at "`eval.rb`", page 44.

<*eval—Main–Process Line-If Not Line*>
This chunk is called by <*eval—Main–Process Line*>; see its first definition at "`eval.rb` Main Process Line Code", page 46.

<*eval—Main–Process Line-Is Line*>
This chunk is called by <*eval—Main–Process Line*>; see its first definition at "`eval.rb` Main Process Line Code", page 46.

<*eval—Main–Process Line-Is Line_Indentation*>
This chunk is called by <*eval—Main–Process Line-Is Line*>; see its first definition at "`eval.rb` If Is Line Code", page 47.

<*eval—Main–Process Line-Is Line_Worth Evaluating?*>
This chunk is called by <*eval—Main–Process Line-Is Line*>; see its first definition at "`eval.rb` If Is Line Code", page 47.

Bibliography

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y