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 0.7a of *Ruby 2.5 Information and Documentation*.

Short Contents

Table of Contents

Preface

Think think think think . . .

> Just remember that in Ruby, `Class` is an object, and `Object` is a class.
>
> —Hal Fulton, *The Ruby Way*

Intended Audience

> The combination of the power of a pure object-oriented language with the convenience of a scripting language makes Ruby a favorite tool of intelligent, forward-thinking programmers.
>
> *Programming Ruby*, by *Dave Thomas* and *Chad Fowler* and *Andy Hunt*

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.

Appendix A "Ruby-Doc", page 104,

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 Ruby Help Tools

The RDoc system includes the command-line tools `rdoc` and `ri` for generating and displaying online documentation.

`ri` (Ruby Index) and RDoc (Ruby Documentation)[1] are a closely related pair of tools for providing documentation about Ruby programs. `ri` and `rdoc` are standalone programs; you run them from the command line. The Ruby `ri` tool is used to view the Ruby documentation off-line. *RDoc* produces HTML and online documentation for Ruby projects.

RDoc

`rdoc`

2.2.1 RDoc

RDoc and the tool `rdoc` are a program documentation system. RDoc produces HTML and command-line documentation for Ruby projects. If you put comments in your program files (Ruby or C) in the prescribed RDoc format, `rdoc` scans your files, extracts the comments, organizes them intelligently (indexed according to what they comment on), and creates nicely formatted documentation from them. You can see RDoc markup in many of the C files in the Ruby source tree and many of the Ruby files in the Ruby installation.

2.2.1.1 Writing Documentation for RDoc

To write documentation for RDoc place a comment above the class, module, method, constant, or attribute you want documented:

```
##
# This class represents an arbitrary shape by a series of points.

class Shape

  ##
  # Creates a new shape described by a +polyline+.
  #
  # If the +polyline+ does not end at the same point it started at the
  # first pointed is copied and placed at the end of the line.
  #
  # An ArgumentError is raised if the line crosses itself, but shapes may
  # be concave.

  def initialize polyline
    # ...
  end
```

[1] Not to be confused with the Ruby Programming Language Documentation at Ruby-Doc (see Appendix A "Ruby-Doc", page 104)

```
end
```

Comment Markup Formats

The default comment markup format is the `RDoc::Markup` format. `TomDoc`, `Markdown` and `RD` format comments are also supported.

Directives

Comments can contain directives that tell RDoc information that it cannot otherwise discover through parsing. See `RDoc::Markup@Directives` to control what is or is not documented, to define method arguments or to break up methods in a class by topic. See `RDoc::Parser::Ruby` for directives used to teach RDoc about metaprogrammed methods.

Documentation Coverage Report

To determine how well your project is documented run `rdoc -C lib` to get a documentation coverage report. `rdoc -C1 lib` includes parameter names in the documentation coverage report.

2.2.1.2 Generating Documention

Create documentation using the `rdoc` command:

```
$ rdoc --help
$ rdoc [options] [names...]
```

A typical use might be to generate documentation for a package of Ruby source. The command `rdoc` generates documentation for all the Ruby and C source files in and below the current directory. These will be stored in a documentation tree starting in the subdirectory `doc`.

You can make this slightly more useful for your readers by having the index page contain the documentation for the primary file.

```
$ rdoc --main README.rdoc
```

To generate documentation programmatically:

```
gem 'rdoc'
require 'rdoc/rdoc'

options = RDoc::Options.new
# see RDoc::Options

rdoc = RDoc::RDoc.new
rdoc.document options
# see RDoc::RDoc
```

2.2.1.3 Markup Directives

Directives are keywords surrounded by `:` characters.

Controlling what is documented

`:nodoc: / :nodoc: all`
: This directive prevents documentation for the element from being generated. For classes and modules, methods, aliases, constants, and attributes directly within the affected class or module also will be omitted. By default, though, modules and classes within that class or module will be documented. This is turned off by adding the all modifier.

Method arguments

`:arg: or :args:` *parameters*
: Overrides the default argument handling with exactly these parameters.

Sections

Sections allow you to group methods in a class into sensible containers. If you use the sections 'Public', 'Internal' and 'Deprecated' (the three allowed method statuses from TomDoc) the sections will be displayed in that order placing the most useful methods at the top. Otherwise, sections will be displayed in alphabetical order.

`:category:` *section*
: Adds this item to the named section overriding the current section. Use this to group methods by section in RDoc output while maintaining a sensible ordering (like alphabetical).

Other directives

`:markup:` *type*
: Overrides the default markup type for this comment with the specified markup type. For Ruby files, if the first comment contains this directive it is applied automatically to all comments in the file.

  Unless you are converting between markup formats you should use a `.rdoc_options` file to specify the default documentation format for your entire project. See Saved Options at `RDoc::Options` for instructions.

2.2.2 `ri` Help System

> The name stands for Ruby Interactive (not to be confused with Interactive Ruby).

The Ruby `ri` tool is used to view the Ruby documentation off-line. Open a command window and invoke `ri` followed by the name of a Ruby class, module or method. `ri` will display documentation for you. You may specify a method name without a qualifying class or module name, but this will just show you a list of all methods by that name (unless the method is unique). Normally, you can separate a class or module name from a method name with a period. If a class defines a class method and an instance method by the same name, you must instead use `::` to refer to a class method or `#` to refer to the instance method. Here are some example invocations of `ri`:

```
ri Array
ri Array.sort
```

```
ri Hash#each
ri Math::sqrt
```

`ri` dovetails with RDoc: It gives you a way to view the information that RDoc has extracted and organized. Specifically (although not exclusively, if you customize it), `ri` is configured to display the RDoc information from the Ruby source files. Thus on any system that has Ruby fully installed, you can get detailed information about Ruby with a simple command-line invocation of `ri`. Some more information is available here:

Classes

To see a list of all the classes for which `ri` has documentation, type the following: *`ri -c`*. Then try accessing the documentation for a sample class: *`ri Hash`*. Next, try viewing the documentation for a given method, for example, one from the `Array` class, rather than `Hash`, just to give as much variety as possible: *`ri 'Array#<<'`*. You will sometimes need to quote the method name, in this case `<<`, to avoid the shell interpreting certain metacharacters.

In one of your shell's start-up files, you may care to alias `ri` to '`ri -f ansi`', which will ensure you get a nice coloured display (as in the example above) when displaying documentation.

2.2.3 The Ruby Debugger

The Ruby debugger is a library loaded into Ruby at run-time. This is done as follows:

```
ruby -r debug [
            options
            ] [
            programfile
            ] [
            arguments
            ]
```

The debugger can do all the usual sorts of things you would expect it to, such as set breakpoints, step into and over code, print out the call stack, etc.

2.2.4 Yard

Yard

Yard — A Ruby Documentation Tool

What Is Yard?

YARD is a documentation generation tool for the Ruby programming language. It enables the user to generate consistent, usable documentation that can be exported to a number of formats very easily, and also supports extending for custom Ruby constructs such as custom class level definitions.

And of course YARD comes with much more functionality, including the ability to serve documentation for gems, the ability to group methods into logical sections, and much more. You can read about all of the new features in YARD 0.6 in the What's New document.

2.2.4.1 Features of Yard

Yardoc Meta-tag Formatting

YARD uses a `@tag` style definition syntax (like Python, Java, Objective-C and other languages) for meta tags alongside regular code documentation. These tags should be able to happily sit side by side RDoc formatted documentation, but provide a much more consistent and usable way to describe important information about objects, such as what parameters they take and what types they are expected to be, what type a method should return, what exceptions it can raise, if it is deprecated, etc.. It also allows information to be better (and more consistently) organized during the output generation phase. You can find a list of tags in the Tags.md file.

YARD also supports an optional *types* declarations for certain tags. This allows the developer to document type signatures for ruby methods and parameters in a non intrusive but helpful and consistent manner. Instead of describing this data in the body of the description, a developer may formally declare the parameter or return type(s) in a single line. Consider the following Yardoc'd method:

```
# Reverses the contents of a String or IO object.
#
# @param [String, #read] contents the contents to reverse
# @return [String] the contents reversed lexically
def reverse(contents)
  contents = contents.read if respond_to? :read
  contents.reverse
end
```

With the above `@param` tag, we learn that the contents parameter can either be a String or any object that responds to the `read` method, which is more powerful than the textual description, which says it should be an IO object. This also informs the developer that they should expect to receive a String object returned by the method, and although this may be obvious for a `reverse` method, it becomes very useful when the method name may not be as descriptive.

RDoc Formatting Compatibility

YARD is made to be compatible with RDoc formatting. In fact, YARD does no processing on RDoc documentation strings, and leaves this up to the output generation tool to decide how to render the documentation.

A Local Documentation Server

YARD can serve documentation for projects or installed gems (similar to gem server) with the added benefit of dynamic searching, as well as live reloading. Using the live reload feature, you can document your code and immediately preview the results by refreshing the page; YARD will do all the work in re-generating the HTML. This makes writing documentation a much faster process.

Custom Constructs and Extensibility

YARD is designed to be extended and customized by plugins. Take for instance the scenario where you need to document the following code:

```
# Sets the publisher name for the list.
```

```
cattr_accessor :publisher
```

This custom declaration provides dynamically generated code that is hard for a documentation tool to properly document without help from the developer. To ease the pains of manually documenting the procedure, YARD can be extended by the developer to handle the `cattr_accessor` construct and automatically create an attribute on the class with the associated documentation. This makes documenting external API's, especially dynamic ones, a lot more consistent for consumption by the users.

YARD is also designed for extensibility everywhere else, allowing you to add support for new programming languages, new data structures and even where/how data is stored.

Template Customization

YARD makes it easy to customize templates using a specially designed templating system. The design allows plugin developers to make small modifications to a template without breaking changes that may have been made from another plugin. This means you can install multiple plugins that each make independent modifications without running into problems with your template. It also allows you to easily make small changes (like adding your own stylesheets) without digging into any markup.

Raw Data Output

YARD also outputs documented objects as raw data (the dumped Namespace) which can be reloaded to do generation at a later date, or even auditing on code. This means that any developer can use the raw data to perform output generation for any custom format, such as YAML, for instance. While YARD plans to support XHTML style documentation output as well as command line (text based) and possibly XML, this may still be useful for those who would like to reap the benefits of YARD's processing in other forms, such as throwing all the documentation into a database. Another useful way of exploiting this raw data format would be to write tools that can auto generate test cases, for example, or show possible unhandled exceptions in code.

2.2.4.2 Yard Guides

Getting Started with YARD

https://www.rubydoc.info/gems/yard/file/docs/GettingStarted.md

Documenting Code with YARD

By default, YARD is compatible with the same RDoc syntax most Ruby developers are already familiar with. However, one of the biggest advantages of YARD is the extended meta-data syntax, commonly known as *tags*, that you can use to express small bits of information in a structured and formal manner. While RDoc syntax expects you to describe your method in a completely free-form manner, YARD recommends declaring your parameters, return types, etc. with the `@tag` syntax, which makes outputting the documentation more consistent and easier to read. Consider the RDoc documentation for a method to_format:

```
# Converts the object into textual markup given a specific ‘format‘
# (defaults to ‘:html‘)
#
```

```
# == Parameters:
# format::
#   A Symbol declaring the format to convert the object to. This
#   can be ‘:text‘ or ‘:html‘.
#
# == Returns:
# A string representing the object in a specified
# format.
#
def to_format(format = :html)
  # format the object
end
```

While this may seem easy enough to read and understand, it's hard for a machine to properly pull this data back out of our documentation. Also we've tied our markup to our content, and now our documentation becomes hard to maintain if we decide later to change our markup style (maybe we don't want the `:` suffix on our headers anymore).

In YARD, we would simply define our method as:

```
# Converts the object into textual markup given a specific format.
#
# @param format [Symbol] the format type, ‘:text‘ or ‘:html‘
# @return [String] the object converted into the expected format.
def to_format(format = :html)
  # format the object
end
```

Using tags we can add semantic metadata to our code without worrying about presentation. YARD will handle presentation for us when we decide to generate documentation later.

Which Markup Format?

YARD does not impose a specific markup. The above example uses standard RDoc markup formatting, but YARD also supports *textile* and *markdown* via the command-line switch or `.yardopts` file (see below). This means that you are free to use whatever formatting you like. This guide is actually written using *markdown*. YARD, however, does add a few important syntaxes that are processed no matter which markup formatting you use, such as tag support and inter-document linking. These syntaxes are discussed below.

Adding Tags to Documentation

The tag syntax that YARD uses is the same `@tag`-style syntax you may have seen if you've ever coded in Java, Python, PHP, Objective-C or a myriad of other languages. The following tag adds an *author* tag to your class:

```
# @author Loren Segal
class MyClass
end
```

To allow for large amounts of text, the `@tag` syntax will recognize any indented lines following a tag as part of the tag data. For example:

```
# @deprecated Use {#my_new_method} instead of this method because
#   it uses a library that is no longer supported in Ruby 1.9.
#   The new method accepts the same parameters.
def mymethod
end
```

List of Tags

A list of tags can be found in Tags.

Reference Tags

To reduce the amount of duplication in writing documentation for repetitive code, YARD introduces *reference tags*, which are not quite tags, but not quite docstrings either. In a sense, they are tag (and docstring) modifiers. Basically, any docstring (or tag) that begins with '`(see OTHEROBJECT)`' will implicitly link the docstring or tag to the '`OTHEROBJECT`', copying any data from that docstring/tag into your current object. Consider the example:

```
class MyWebServer
  # Handles a request
  # @param request [Request] the request object
  # @return [String] the resulting webpage
  def get(request) "hello" end

  # (see #get)
  def post(request) "hello" end
end
```

The above `#post` method takes the docstring and all tags (param and return) of the `#get` method. When you generate HTML documentation, you will see this duplication automatically, so you don't have to manually type it out. We can also add our own custom docstring information below the '`see`' reference, and whatever we write will be appended to the docstring:

```
# (see #get)
# @note This method may modify our application state!
def post(request) self.state += 1; "hello" end
```

Here we added another tag, but we could have also added plain text. The text must be appended after the '`(see ...)`' statement, preferably on a separate line.

Note that we don't have to "refer" the whole docstring. We can also link individual tags instead. Since `get` and `post` actually have different descriptions, a more accurate example would be to only refer our parameter and return tags:

```
class MyWebServer
  # Handles a GET request
  # @param request [Request] the request object
  # @return [String] the resulting webpage
  def get(request) "hello" end

  # Handles a POST request
  # @note This method may modify our application state!
```

```
  # @param (see #get)
  # @return (see #get)
  def post(request) self.state += 1; "hello" end
end
```

The above copies all of the param and return tags from `#get`. Note that you cannot copy individual tags of a specific type with this syntax.

Declaring Types

Some tags also have an optional "types" field which let us declare a list of types associated with the tag. For instance, a return tag can be declared with or without a types field.

```
# @return [String, nil] the contents of our object or nil
#   if the object has not been filled with data.
def validate; end

# We don't care about the "type" here:
# @return the object
def to_obj; end
```

The list of types is in the form '`[type1, type2, ...]`' and is mostly free-form, so we can also specify duck-types or constant values. For example:

```
# @param argname [#to_s] any object that responds to `#to_s`
# @param argname [true, false] only true or false
```

Note the latter example can be replaced by the meta-type "Boolean". Another meta-type is "void", which stands for "no meaningful value" and is used for return values. These meta-types are by convention only, but are recommended.

List types can be specified in the form `CollectionClass<ElementType, ...>`. For instance, consider the following Array that holds a set of Strings and Symbols:

```
# @param list [Array<String, Symbol>] the list of strings and symbols.
```

We mentioned that these type fields are "mostly" free-form. In truth, they are defined "by convention". To view samples of common type specifications and recommended conventions for writing type specifications, see http://yardoc.org/types.html. Note that these conventions may change every now and then, although we are working on a more "formal" type specification proposal.

Documenting Attributes

To document a Ruby attribute, add documentation text above the attribute definition.

```
# Controls the amplitude of the waveform.
# @return [Numeric] the amplitude of the waveform
attr_accessor :amplitude
```

As a short-hand syntax for declaring reader and writer attribute pairs, YARD will automatically wire up the correct method types and information by simply defining documentation in the `@return` tag. For example, the following declaration will show the correct information for the waveform attribute, both for the getter's return type and the setter's value parameter type:

```
# @return [Numeric] the amplitude of the waveform
```

```
attr_accessor :amplitude
```

In this case, the most important details for the attribute are the object type declaration and its descriptive text.

Documentation for a Separate Attribute Writer

Usually an attribute will get and set a value using the same syntax, so there is no reason to have separate documentation for an attribute writer. In the above amplitude case, the Numeric type is both used for the getter and setter types.

Sometimes, however, you might want to have separate documentation for the getter and setter. In this case, you would still add the documentation text to the getter declaration (or attr_accessor) and use `@overload` tags to declare the separate docstrings. For example:

```
# @overload amplitude
#   Gets the current waveform amplitude.
#   @return [Numeric] the amplitude of the waveform
# @overload amplitude=(value)
#   Sets the new amplitude.
#   @param value [Numeric] the new amplitude value
#   @note The new amplitude will only take effect if {#restart}
#     is called on the stream.
```

Note that by default, YARD exposes the reader portion of the attribute in HTML output. If you have separate `attr_reader` and `attr_writer` declarations, make sure to put your documentation (for both reader and writer methods) on the reader declaration using `@overload` tags as described above. For example:

```
# @overload ...documentation here...
attr_reader :amplitude

# This documentation will be ignored by YARD.
attr_writer :amplitude
```

Documenting Custom DSL Methods

. . . As of version 0.7.0, YARD will automatically pick up on these basic methods if you document them with a docstring. Therefore, simply adding some comments to the code will cause it to generate documentation: . . .

Macros

Fortunately YARD 0.7.0 also adds macros, a powerful way to add support for these DSL methods on the fly without writing extra plugins. Macros allow you to interpolate arguments from the method call inside the docstring, reducing duplication. If we re-wrote the property example from above using a macro, it might look like: . . .

Customized YARD Markup

YARD supports a special syntax to link to other code objects, URLs, files, or embed docstrings between documents. This syntax has the general form of '`{Name OptionalTitle}`' (where '`OptionalTitle`' can have spaces, but '`Name`' cannot).

Linking Objects '`{...}`'

To link another "object" (class, method, module, etc.), use the format: ...

Linking URLs '`{http://...}`'

URLs are also linked using this '`{...}`' syntax: ...

Linking Files '`{file:...}`'

Files can also be linked using this same syntax but by adding the `file:` prefix to the object name. Files refer to extra readme files you added via the command-line. Consider the following examples: ...

Embedding Docstrings '`{include:...}`'

We saw the '`(see ...)`' syntax above, which allowed us to link an entire docstring with another. Sometimes, however, we just want to copy docstring text without tags. Using the same '`{...}`' syntax, but using the `include:` prefix, we can embed a docstring (minus tags) at a specific point in the text. ...

Embedding Files '`{include:file:...}`'

You can embed the contents of files using `{include:file:path/to/file}`, similar to the `{include:OBJECT}` tag above. If the file uses a specific markup type, it will be applied and embedded as marked up text. The following shows how the tag can be used inside of comments: ...

Rendering Objects '`{render:...}`'

Entire objects can also be rendered in place in documentation. This can be used for guide-style documentation which does not document the entire source tree, but instead selectively renders important classes or methods. Consider the following documentation inside of a README file: ...

2.2.4.3 Using YARD to Generate Documentation

`yard` Executable

YARD ships with a single executable aptly named `yard`. In addition to generating standard documentation for your project, you would use this tool if you wanted to:

- Document all installed gems
- Run a local documentation server
- Generate UML diagrams using Graphviz
- View ri-style documentation
- Diff your documentation
- Diff your documentation

`yard` Commands

The following commands are available in YARD 0.6.x (see `yard help` for a full list):

```
Usage: yard <command> [options]
```

```
Commands:
config   Views or edits current global configuration
diff     Returns the object diff of two gems or .yardoc files
doc      Generates documentation
gems     Builds YARD index for gems
graph    Graphs class diagram using Graphviz
help     Retrieves help for a command
ri       A tool to view documentation in the console like `ri`
server   Runs a local documentation server
stats    Prints documentation statistics on a set of files
```

Note that `yardoc` is an alias for `yard doc`, and `yri` is an alias for `yard ri`. These commands are maintained for backwards compatibility.

`.yardopts` Options File

Unless your documentation is very small, you'll end up needing to run `yardoc` with many options. The `yardoc` tool will use the options found in this file. It is recommended to check this in to your repository and distribute it with your source. This file is placed at the root of your project (in the directory you run `yardoc` from) and contains all of arguments you would otherwise pass to the command-line tool. For instance, if you often type:

```
yardoc --no-private --protected app/**/*.rb - README LEGAL COPYING
```

You can place the following into your `.yardopts`:

```
--no-private --protected app/**/*.rb - README LEGAL COPYING
```

This way, you only need to type: `yardoc`. Any extra switches passed to the command-line now will be appended to your `.yardopts` options.

Note that options for `yardoc` are discussed in the `README`, and a full overview of the `.yardopts` file can be found in `YARD::CLI::Yardoc`.

Documenting Extra Files

Extra files are extra guide style documents that help to give a brief overview of how to use your library/framework, as well as any extra information that might be vital for your users. The most common *extra file* is the `README`, which is automatically detected by YARD if found in the root of your project (any file starting with `README*`). You can specify extra files on the command line (or in the `.yardopts` file) by listing them after the `-` separator:

```
yardoc lib/**/*.rb ext/**/*.c - LICENSE.txt
```

Note that the `README` will automatically be picked up, so you do not need to specify it. If you don't want to modify the default file globs, you can ignore the first set of arguments:

```
yardoc - LICENSE.txt
```

Below you can read about how to customize the look of these extra files, both with markup and pretty titles.

Adding Meta-Data to Extra Files

You can add YARD-style `@tag` metadata to the top of any extra file if prefixed by a # hash comment. YARD allows for arbitrary meta-data, but pays special attention to the

tags `@markup`, `@encoding`, and `@title`. Note that there cannot be any whitespace before the tags. Here is an example of some tag data in a README:

```
# @markup markdown
# @title The Best Library in the World!
# @author The Author Name

This is the best library you will ever meet. Lipsum ...
```

The `@markup` tag allows you to specify a markup format to use for the file, including "markdown", "textile", "rdoc", "ruby", "text", "html", or "none" (no markup). This can be used when the markup cannot be auto-detected using the extension of the filename, if the file has no extension, or if you want to override the auto-detection.

By using `@encoding` you can specify a non-standard encoding. . . .

The `@title` tag allows you to specify a full title name for the document. By default, YARD uses the filename as the title of the document and lists it in the file list in the index and file menu. In some cases, the file name might not be descriptive enough, so YARD allows you to specify a full title:

```
contents of TITLE.txt:
# @title The Title of The Document
```

. . .

2.2.4.4 Configuring YARD

YARD (0.6.2+) supports a global configuration file stored in `~/.yard/config`. This file is stored as a YAML file and can contain arbitrary keys and values that can be used by YARD at run-time. YARD defines specific keys that are used to control various features, and they are listed in `YARD::Config::DEFAULT_CONFIG_OPTIONS`. A sample configuration file might look like:

```
:load_plugins: false
:ignored_plugins:
  - my_plugin
  - my_other_plugin
:autoload_plugins:
  - my_autoload_plugin
:safe_mode: false
```

You can also view and edit these configuration options from the commandline using the yard `config` command. To list your configuration, use '`yard config --list`'. To view a key, use '`yard config ITEM`', and to set it, use '`yard config ITEM VALUE`'.

Extending YARD

. . .

Templating YARD

. . .

Plugin Support

. . .

2.2.4.5 Yard API Documentation

https://www.rubydoc.info/gems/yard

YARD is a documentation generation tool for the Ruby programming language. It enables the user to generate consistent, usable documentation that can be exported to a number of formats very easily, and also supports extending for custom Ruby constructs such as custom class level definitions.

Installing

To install YARD, use the following command:

```
$ gem install yard
```

Alternatively, if you've checked the source out directly, you can call `rake install` from the root project directory.

Usage

There are a couple of ways to use YARD. The first is via command-line, and the second is the Rake task.

1. `yard` Command-line Tool

   YARD comes packaged with an executable named `yard` which can control the many functions of YARD, including generating documentation, graphs running the YARD server, and so on. To view a list of available YARD commands, type:

   ```
   $ yard --help
   ```

   ### Generating Documentation

   The `yardoc` executable is a shortcut for `yard doc`.

   The most common command you will probably use is `yard doc`, or `yardoc`. You can type `yardoc --help` to see the options that YARD provides, but the easiest way to generate docs for your code is to simply type `yardoc` in your project root. This will assume your files are located in the `lib/` directory. If they are located elsewhere, you can specify paths and globs from the commandline via:

   ```
   $ yardoc 'lib/**/*.rb' 'app/**/*.rb' ...etc...
   ```

   The tool will generate a `.yardoc` file which will store the cached database of your source code and documentation. If you want to re-generate your docs with another template you can simply use the `--use-cache` (or `-c`) option to speed up the generation process by skipping source parsing.

   YARD will by default only document code in your public visibility. You can document your protected and private code by adding `--protected` or `--private` to the option switches. In addition, you can add `--no-private` to also ignore any object that has the `@private` meta-tag. This is similar to RDoc's `:nodoc:` behaviour, though the distinction is important. RDoc implies that the object with `:nodoc:` would not be

documented, whereas YARD still recommends documenting private objects for the private API (for maintainer/developer consumption).

You can also add extra informative files (`README`, `LICENSE`) by separating the globs and the filenames with `-`.

```
$ yardoc 'app/**/*.rb' - README LICENSE FAQ
```

If no globs precede the `-` argument, the default glob ('`lib/**/*.rb`') is used:

```
$ yardoc - README LICENSE FAQ
```

Note that the README file can be specified with its own `--readme` switch.

You can also add a `.yardopts` file to your project directory which lists the switches separated by whitespace (newlines or space) to pass to `yardoc` whenever it is run. A full overview of the `.yardopts` file can be found in `YARD::CLI::Yardoc`.

Queries

The yardoc tool also supports a `--query` argument to only include objects that match a certain data or meta-data query. The query syntax is Ruby, though a few shortcuts are available. For instance, to document only objects that have an `@api` tag with the value `public`, all of the following syntaxes would give the same result:

```
--query '@api.text == "public"'
--query 'object.has_tag?(:api) && object.tag(:api).text == "public"'
--query 'has_tag?(:api) && tag(:api).text == "public"'
```

...

2. Rake Task

   ...

3. `yri` RI Implementation

   ...

4. `yard server` Documentation Server

   The `yard server` command serves documentation for a local project or all installed RubyGems. To serve documentation for a project you are working on, simply run:

   ```
   $ yard server
   ```

   And the project inside the current directory will be parsed (if the source has not yet been scanned by YARD) and served at '`http://localhost:8808`'.

Live Reloading

If you want to serve documentation on a project while you document it so that you can preview the results, simply pass `--reload` (`-r`) to the above command and YARD will reload any changed files on each request. This will allow you to change any documentation in the source and refresh to see the new contents.

Serving Gems

To serve documentation for all installed gems, call:

```
$ yard server --gems
```

This will also automatically build documentation for any gems that have not been previously scanned. Note that in this case there will be a slight delay between the first request of a newly parsed gem.

5. `yard graph` Graphviz Generator

   . . .

2.3 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 22,
- "Developing Ruby", page 22,
- "Patch by Patch", page 22,
- 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

2.4 Getting Started

2.4.1 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.

2.4.2 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 2.4.2.1 “FAQ Iterators”, page 23,
- Section 2.4.2.2 “FAQ Syntax”, page 25,
- Methods
- Classes and modules
- Built-in libraries
- Extension library
- Other features

2.4.2.1 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

2.4.2.2 FAQ Syntax

List of FAQ items:

"FAQ Syntax", page 25,
"FAQ Syntax", page 26,
"FAQ Syntax", page 26,
"FAQ Syntax", page 26,
"FAQ Syntax", page 26,
"FAQ Syntax", page 27,
"FAQ Syntax", page 27,
"FAQ Syntax", page 27,
"FAQ Syntax", page 27,
"FAQ Syntax", page 28,
"FAQ Syntax", page 28,
"FAQ Syntax", page 28,
"FAQ Syntax", page 28,
"FAQ Syntax", page 28,
"FAQ Syntax", page 29,

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`).

2.4.2.3 FAQ Methods

How does Ruby choose which method to invoke?

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

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

What is a singleton method?

A *singleton method* is an instance method associated with one specific object. You create a singleton method by including the object in the definition:

```
class Foo; end

foo = Foo.new
bar = Foo.new

def foo.hello
  puts "Hello"
end
```

```
foo.hello
⇒Hello

bar.hello
⇒ prog.rb:11:in `<main>': undefined method `hello' for
#<Foo:0x000000010f5a40> (NoMethodError)
```

See "FAQ Classes and Modules", page 30.

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?

2.4.2.4 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?

A singleton method of a class object is called a *class method* (see "FAQ Methods", page 29). (Actually, the class method is defined in the metaclass, but that is pretty much transparent).

Another way of looking at it is to say that *a class method is a method whose receiver is a class.*

It all comes down to the fact that you can call class methods without having to have instances of that class (objects) as the receiver.

```
class Foo
  def self.test
    "this is foo"
  end
end

# It is invoked this way.

Foo.test  # => "this is foo"
```

In this example, `Foo.test` is a class method.

Instance methods which are defined in class `Class` can be used as class methods for every(!) class.

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?

You can define a class method in the class definition, and you can define a class method at the top level.

```
class Demo
  def self.class_method
  end
end

def Demo.another_class_method
end
```

There is only one significant difference between the two. In the class definition you can refer to the classs constants directly, as the constants are within scope. At the top level, you have to use the `Class::CONST` notation.

What is the difference between include and extend?

What does `self` mean?

2.4.2.5 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`?

2.4.2.6 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?

2.4.2.7 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?

2.4.3 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.

2.4.4 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.

2.4.5 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.)

2.4.5.1 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.

2.4.5.2 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`.

2.4.5.3 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.

2.4.5.4 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 38.
<*MegaGreeter—Main Script*>	See "Large Class Definition", page 40.
<*MegaGreeter—say_bye Method*>	See "Large Class Definition", page 39.
<*MegaGreeter—say_hi Method*>	See "Large Class Definition", page 38.

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 37.

`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 37.

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 37.

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 37.

2.4.5.5 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!
...
...
```

2.4.6 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.

2.4.6.1 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.

2.4.6.2 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). See Section 4.6 "Equality", page 97,

2.4.6.3 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"`).

2.4.6.4 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).

2.4.6.5 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.

2.4.7 Important Language Features

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

2.4.7.1 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`).

2.4.7.2 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
```

2.4.7.3 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`).

2.4.7.4 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
```

2.4.7.5 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.

2.4.7.6 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.

2.4.7.7 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."
```

2.4.7.8 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`'.

2.4.7.9 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.

2.4.7.10 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>
```

2.4.7.11 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
```

2.4.7.12 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.

2.4.7.13 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
```

2.4.7.14 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.

2.4.7.15 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
```

2.4.7.16 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!
```

2.4.7.17 Operators are syntactic sugar

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

2.4.8 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.

2.4.9 Ruby Essentials

Ruby Essentials

Ruby Essentials is a free on-line book designed to provide a concise and easy to follow [sic] guide to learning Ruby.

2.4.9.1 Interactive Ruby Execution

Interactive Ruby code is entered using the `irb` tool.

Once irb is installed, launch it as follows:

```
$ irb
irb(main):001:0>
```

Now, we can begin to execute Ruby code:

```
irb(main):001:0> puts 'Hello Ruby'
Hello Ruby
=> nil
irb(main):002:0>
```

2.4.9.2 Block Ruby Commenets

Multiple lines of text or code can be defined as comments using the Ruby `=begin` and `=end` comment markers. These are known as the *comment block markers*.

2.4.9.3 Variable Scope

Scope defines where in a program a variable is accessible. Ruby has four types of variable scope, plus one constant type. Each variable type is declared by using a special character at the start of the variable name as outlined in the following table.

local	[a-z] or _
global	`$`
instance	`@`
class	`@@`
constant	[A-Z]

Detecting The Scope Of A Variable

Sometimes you need to find out the scope programmatically. A useful technique to find out the scope of a variable is to use the `defined?` method. `defined?` will return the scope of the variable referenced, or `nil` if the variable is not defined in the current context.

```
x = 10
=> 10
defined? x
=> "local-variable"

$x = 10
=> 10
defined? $x
=> "global-variable"
```

Predefined Global Variables

See "Files API", page 105,

`$@`	The location of latest error
`$_`	The string last read by gets
`$.`	The line number last read by interpreter
`$&`	The string last matched by regexp
`$~`	The last regexp match, as an array of subexpressions
`$n`	The nth subexpression in the last match (same as $~[n])
`$=`	The case-insensitivity flag
`$/`	The input record separator
`$\`	The output record separator
`$0`	The name of the ruby script file currently executing

`$*` The command line arguments used to invoke the script

`$$` The Ruby interpreter's process ID

`$?` The exit status of last executed child process

2.4.10 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

2.5 Manuals

Manuals

2.5.1 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

2.5.1.1 On What Ruby Is

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

2.5.1.2 On 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.[2]

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 language 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
```

[2] Ruby does not convert strings into integers automatically like perl does.

```
1
% ruby fact.rb 5
120

% ruby fact.rb 40
815915283247897734345611269596115894272000000000

% ruby fact.rb 400
64034522846623895262347970319503005850702583026002959458684
44594280239716918683143627847864746326467629435057503585681
08482981628835174352289619886468029979373416541508381624264
61942352307046244325015114448670890662773914918117331955996
44070954967134529047702032243491121079759328079510154537266
72516278778900093497637657103263503315339653498683868313393
52024373788157786791506311858702618270169819740062983025308
59129834616227230455833952075961150530223608681043329725519
48526744322324386699484224042325990555161063594237696139923
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` (see Section D.1 "Ruby Eval Utility", page 215) 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.

2.5.1.3 On Strings

Quoting Strings

A string may be double-quoted ('`"..."`') or single-quoted ('`'...'`').

Double- and single-quoting have different effects in some cases. A double-quoted string allows character escapes by a leading backslash, and the evaluation of embedded expressions using `#{}`. A single-quoted string does not do this interpreting; what you see is what you get.

String Methods

You can concatenate strings with `+`, and repeat a string many times with `*`.

Here are some things you can do with strings.

Concatenation
'`word = "fo" + "o"`' ⇒ '`"foo"`'

Repetition '`word = word * 2`' ⇒ '`"foofoo"`'

Extracting characters
(note that characters are integers in ruby)
'`word[0]`' ⇒ '`102`' '`# 102 is ASCII code of 'f'`'
'`word[-1]`' ⇒ '`111`' '`# 111 is ASCII code of 'o'`'

Extracting substrings

```
ruby> herb = "parsley"
   "parsley"
ruby> herb[0,1]
   "p"
ruby> herb[-2,2]
   "ey"
ruby> herb[0..3]
   "pars"
ruby> herb[-5..-2]
   "rsle"
```

Testing for equality:
'`"foo" == "foo"`' ⇒ '`true`'
'`"foo" == "bar"`' ⇒ '`false`'

2.5.1.4 On Puzzle Program

{guess.rb} ≡

```
# save this as guess.rb
words = ['foobar', 'baz', 'quux']
secret = words[rand(3)]

print "guess? "
while guess = STDIN.gets
  guess.chop!
  if guess == secret
    puts "You win!"
    break
  else
    puts "Sorry, you lose."
  end
  print "guess? "
end
puts "The word was ", secret, "."
```

New Control Structure `while`

In this program, a new control structure, `while`, is used. The code between `while` and its corresponding `end` will execute repeatedly as long as some specified condition remains true. In this case, '`guess=STDIN.gets`' is both an active statement (collecting a line of user input and storing it as guess), and a condition (if there is no input, guess, which repesents the value of the whole '`guess=STDIN.gets`' expression, has a `nil` value, causing while to stop looping).

Standard Input Object — *STDIN*

STDIN is the standard input object. Usually, '`guess=gets`' does the same thing as '`guess=STDIN.gets`'. In line 5 we read one line from standard input by the method *STDIN*.`gets`. If *EOF* (end of file) occurs while getting the line, `gets` returns `nil`. So the code associated with this `while` will repeat until it sees `^D` signifying the end of input.

`guess.chop!`

`guess.chop!` in line 6 deletes the last character from `guess`; in this case it will always be a `newline` character, `gets` includes that character to reflect the user's `return` keystroke, but we're not interested in it.

Printing Variables

In line 15 we print the secret word. We have written this as a `puts` (`put s`tring) statement with two arguments, which are printed one after the other; but it would have been equally effective to do it with a single argument, writing secret as `#{secret}` to make it clear that it is a variable to be evaluated, not a literal word to be printed:

```
puts "the word is #{secret}."
```

It builds a single string and presents it as a single argument to `puts`.

`print` vs `puts`

Also, we are by now used to the idea of using `puts` for standard script output, but this script uses `print` instead, in lines 4 and 13. They are not quite the same thing. `print` outputs exactly what it is given; `puts` also ensures that the output line ends. Using `print` in lines 4 and 13 leaves the cursor next to what was just printed, rather than moving it to the beginning of the next line. This creates a recognizable prompt for user input. In general, the four output calls below are equivalent:

Flushing Standard Output

Sometimes a text window is programmed to *buffer* output for the sake of speed, collecting individual characters and displaying them only when it is given a `newline` character. So if the guessing game script misbehaves by not showing the prompt lines until after the user supplies a guess, *buffering* is the likely culprit. To make sure this doesn't happen, you can "flush" the output as soon as you have printed the prompt. It tells the standard output device (an object named *STDOUT*), "don't wait; display what you have in your buffer right now." '`04 print "guess? "; STDOUT.flush`'.

2.5.1.5 Regular Expressions

```
ruby> def chab(s)   # "contains hex in angle brackets"
    |    (s =~ /<0(x|X)(\d|[a-f]|[A-F])+>/) != nil
    | end
  nil
ruby> chab "Not this one."
  false
ruby> chab "Maybe this? {0x35}"    # wrong kind of brackets
  false
ruby> chab "Or this? <0x38z7e>"    # bogus hex digit
  false
ruby> chab "Okay, this: <0xfc0004>."
  true
```

Program To Help Experiment With Regular Expressions

Here is a little program to help you experiment with regular expressions. Store it as `regx.rb` and run it by typing *`ruby regx.rb`* at the command line.

The program requires input twice, once for a string and once for a regular expression. The string is tested against the regular expression, then displayed with all the matching parts highlighted in reverse video.

{`regx.rb`} ≡

```
# Requires an ANSI terminal!

st = "\033[7m"
en = "\033[m"

puts "Enter an empty string at any time to exit."
```

```
while true
  print "str> "; STDOUT.flush; str = gets.chop
  break if str.empty?
  print "pat> "; STDOUT.flush; pat = gets.chop
  break if pat.empty?
  re = Regexp.new(pat)
  puts str.gsub(re,"#{st}\\&#{en}")
end
```

Explication of `regx.rb`

The `break` Statement

In line 6, the condition for `while` is hardwired to `true`, so it forms what looks like an infinite loop. However we put `break` statements in the 8th and 10th lines to escape the loop.

`if` Modifiers

```
break if str.empty?
break if pat.empty?
```

These two `break`s are also an example of *if modifiers*. An `if` modifier executes the statement on its left hand side if and only if the specified condition is satisfied. This construction is unusual in that it operates logically from right to left, but it is provided because for many people it mimics a similar pattern in natural speech. It also has the advantage of brevity, as it needs no `end` statement to tell the interpreter how much of the following code is supposed to be conditional. An `if` modifier is conventionally used in situations where a statement and condition are short enough to fit comfortably together on one script line.

Note the difference in the user interface compared to the string-guessing script. This one lets the user quit by hitting the `Return` key on an empty line. We are testing for emptiness of the input string, not for its nonexistence.

Nondestructive `chops` vs Destructive `chops!`

In lines 7 and 9 we have a *non-destructive* `chop`; again, we're getting rid of the unwanted newline character we always get from `gets`. Add the exclamation point, and we have a *destructive* chop. What's the difference? In ruby, we conventionally attach `!` or `?` to the end of certain method names. The exclamation point (`!`, sometimes pronounced aloud as "bang!") indicates something potentially destructive, that is to say, something that can change the value of what it touches. `chop!` affects a string directly, but `chop` gives you a chopped copy without damaging the original.

`chomp` And `chomp!`

You'll also sometimes see `chomp` and `chomp!` used. These are more selective: the end of a string gets bit off only if it happens to be a newline. So for example, `"XYZ".chomp!` does nothing. If you need a trick to remember the difference, think of a person or animal tasting something before deciding to take a bite, as opposed to an axe chopping indiscriminately.

Predicate Method Naming Convention

The other method naming convention appears in lines 8 and 10. A question mark (`?`, sometimes pronounced aloud as "huh?") indicates a *predicate* method, one that can return either true or false.

Regular Expressions At Work

Line 11 creates a regular expression object out of the string supplied by the user. The real work is finally done in line 12, which uses `gsub` to globally substitute each match of that expression with itself, but surrounded by ansi markups; also the same line outputs the results.

In line 12 we see `\\&`. This is a little tricky. Since the replacement string is in double quotes, the pair of backslashes will be interpreted as a single backslash; what `gsub` actually sees will be `\&`, and that happens to be a special code that refers to whatever matched the pattern in the first place. So the new string, when displayed, looks just like the old one, except that the parts that matched the given pattern are highlighted in inverse video.

The =~ Matching Operator

`=~` is a matching operator with respect to regular expressions; it returns the position in a string where a match was found, or `nil` if the pattern did not match.

```
ruby> "abcdef" =~ /d/
   3
ruby> "aaaaaa" =~ /d/
   nil
```

2.5.1.6 On Arrays And Hashes

Creating An Array

You can create an array by listing some items within square brackets (`[]`) and separating them with commas. Ruby's arrays can accomodate diverse object types. '`ary = [1, 2, "3"]`' ⇒ '`[1, 2, "3"]`'

Concatenating and Repeating Arrays

Arrays can be concatenated or repeated just as strings can. '`ary + ["foo", "bar"]`' ⇒ '`[1, 2, "3", "foo", "bar"]`'; '`ary * 2`' ⇒ '`[1, 2, "3", 1, 2, "3"]`'.

Referring To Elements of Arrays

We can use index numbers to refer to any part of a array. '`ary[0]`' ⇒ '1'; '`ary[0,2]`' ⇒ '`[1, 2]`'; '`ary[0..1]`' ⇒ '`[1, 2]`'.

Converted To And From Strings

Arrays can be converted to and from strings, using `join` and `split` respecitvely. '`str = ary.join(":")`' ⇒ '`"1:2:3"`'. '`str.split(":")`' ⇒ '`["1", "2", "3"]`'.

Hashes

An *associative array* has elements that are accessed not by sequential index numbers, but by keys which can have any sort of value. Such an array is sometimes called a *hash* or

dictionary; in the ruby world, we prefer the term hash. A hash can be constructed by quoting pairs of items within curly braces (`{}`). You use a key to find something in a hash, much as you use an index to find something in an array.

```
ruby> h = {1 => 2, "2" => "4"}
   {1=>2, "2"=>"4"}
ruby> h[1]
   2
ruby> h["2"]
   "4"
ruby> h[5]
   nil
ruby> h[5] = 10    # appending an entry
   10
ruby> h
   {5=>10, 1=>2, "2"=>"4"}
ruby> h.delete 1   # deleting an entry by key
   2
ruby> h[1]
   nil
ruby> h
   {5=>10, "2"=>"4"}
```

2.5.1.7 On Control Structures

The `case` Statement

We use the `case` statement to test a sequence of conditions. This is superficially similar to `switch` in C and Java but is considerably more powerful, as we shall see.

```
ruby> i=8
ruby> case i
    | when 1, 2..5
    |   puts "1..5"
    | when 6..10
    |   puts "6..10"
    | end
6..10
   nil
```

Testing For A Range Of Values

`2..5` is an expression which means the *range* between 2 and 5, inclusive. The following expression tests whether the value of `i` falls within that range: '`(2..5) === i`'.

The Relationship Operator

`case` internally uses the *relationship* operator `===` to check for several conditions at a time. In keeping with ruby's object oriented nature, `===` is interpreted suitably for the object that appeared in the `when` condition.

For example, the following code tests string equality in the first `when`, and regular expression matching in the second `when`.

```
ruby> case 'abcdef'
    | when 'aaa', 'bbb'
    |   puts "aaa or bbb"
    | when /def/
    |   puts "includes /def/"
    | end
includes /def/
   nil
```

The `while` Statement

Ruby provides convenient ways to construct loops, although you will find in the next chapter that learning how to use iterators will make it unnecessary to write explicit loops very often.

A `while` is a repeated `if`. We used it in our word-guessing puzzle and in the regular expression programs (see the previous chapter); there, it took the form '`while condition ... end`' surrounding a block of code to be repeated while condition was true. But `while` and `if` can as easily be applied to individual statements: '`puts "It's zero." if i==0`' ⇒ '`It's zero.`' and '`puts i+=1 while i<3`' ⇒ '`1 2 3`'.

Negated Conditions

Sometimes you want to negate a test condition. An `unless` is a negated `if`, and an `until` is a negated `while`.

Interrupting A Loop

There are four ways to interrupt the progress of a loop from inside.

1. First, `break` means, as in C, to escape from the loop entirely.
2. Second, `next` skips to the beginning of the next iteration of the loop (corresponding to C's continue).
3. Third, ruby has `redo`, which restarts the current iteration.
4. The fourth way to get out of a loop from the inside is `return`. An evaluation of `return` causes escape not only from a loop but from the method that contains the loop. If an argument is given, it will be returned from the method call, otherwise `nil` is returned.

The following is C code illustrating the meanings of `break`, `next`, and `redo`:

```
while (condition) {
label_redo:
   goto label_next;        /* ruby's "next" */
   goto label_break;       /* ruby's "break" */
   goto label_redo;        /* ruby's "redo" */
   ...
   ...
label_next:
}
label_break:
...
```

The `for` Statement

C programmers will be wondering by now how to make a `for` loop. Ruby's `for` can serve the same purpose, but adds some flexibility. The loop below runs once for each element in a collection (array, hash, numeric sequence, etc.), but doesn't make the programmer think about indices:

```
for elt in collection
  # ... here, elt refers to an element of the collection
end
```

The collection can be a range of values (this is what most people mean when they talk about a `for` loop):

```
ruby> for num in (4..6)
    |    puts num
    | end
4
5
6
   4..6
```

`for` Equivalent To `each`

But we're getting ahead of ourselves. `for` is really another way of writing `each`, which, it so happens, is our first example of an iterator. The following two forms are equivalent:

```
#  If you're used to C or Java, you might prefer this.
for element in collection
  ...
end

#  A Smalltalk programmer might prefer this.
collection.each {|element|
  ...
}
```

Iterators can often be substituted for conventional loops, and once you get used to them, they are generally easier to deal with.

2.5.1.8 Ruby User's Guide On Iterators

Iterators are not an original concept with ruby. They are in common use in object-oriented languages. They are also used in Lisp, though there they are not called iterators. However the concept of iterator is an unfamiliar one for many so it should be explained in more detail.

An *iterator* is something that does the same thing many times.

Ruby Allows Us To Define Iterators

So every OOP language includes some facilities for iteration. Some languages provide a special class for this purpose; ruby allows us to define iterators directly.

Iterators In String

Ruby's String type has some useful iterators:

`each_byte` is an iterator for each character in the string. Each character is substituted into the local variable c: '`"abc".each_byte{|c| printf "<%c>", c}; print "\n"`'.

The `each_byte` iterator is both conceptually simpler and more likely to continue to work even if the String class happens to be radically modified in the future. One benefit of iterators is that they tend to be robust in the face of such changes; indeed that is a characteristic of good code in general.

Another iterator of String is `each_line`: '`"a\nb\nc\n".each_line{|l| print l}`'.

The tasks that would take most of the programming effort in C (finding line delimiters, generating substrings, etc.) are easily tackled using iterators.

The `for` statement appearing in the previous chapter does iteration by way of an `each` iterator. String's `each` works the same as `each_line`, so let's rewrite the above example with `for`:

```
ruby> for l in "a\nb\nc\n"
    |   print l
    | end
a
b
c
   nil
```

Control Structures `retry` And `redo`

We can use a control structure `retry` in conjunction with an iterated loop, and it will retry the loop from the beginning. `redo` causes just the current iteration of the loop to be redone.

`yield` In Iterators

`yield` occurs sometimes in a definition of an iterator. `yield` moves control to the block of code that is passed to the iterator (this will be explored in more detail in the chapter about procedure objects).

The following example defines an iterator `repeat`, which repeats a block of code the number of times specified in an argument.

```
ruby> def repeat(num)
    |   while num > 0
    |     yield
    |     num -= 1
    |   end
    | end
   nil
ruby> repeat(3) { puts "foo" }
foo
foo
foo
   nil
```

With `retry`, one can define an iterator which works something like ruby's standard `while`.

```
ruby> def WHILE(cond)
    |   return if not cond
    |   yield
    |   retry
    | end
   nil
ruby> i=0; WHILE(i<3) { print i; i+=1 }
012   nil
```

Summary On Iterarors

There are a few restrictions, but you can write your original iterators; and in fact, whenever you define a new data type, it is often convenient to define suitable iterators to go with it. In this sense, the above examples are not terribly useful. We can talk about practical iterators after we have a better understanding of what classes are.

2.5.1.9 On Object-Oriented Thinking

Ruby claims to be an object oriented scripting language; but what exactly does *object oriented* mean? Rather than sum it too quickly, let's think for a moment about the traditional programming paradigm.

Traditionally, a programming problem is attacked by coming up with some kinds of *data representations*, and *procedures* that operate on that data. Under this model, data is inert, passive, and helpless; it sits at the complete mercy of a large procedural body, which is active, logical, and all-powerful.

The problem with this approach is that programs are written by programmers, who are only human and can only keep so much detail clear in their heads at any one time. As a project gets larger, its procedural core grows to the point where it is difficult to remember how the whole thing works. Minor lapses of thinking and typographical errors become more likely to result in well-concealed bugs. Complex and unintended interactions begin to emerge within the procedural core, and maintaining it becomes like trying to carry around an angry squid without letting any tentacles touch your face. There are guidelines for programming that can help to minimize and localize bugs within this traditional paradigm, but there is a better solution that involves fundamentally changing the way we work.

What object-oriented programming does is to let us delegate most of the mundane and repetitive logical work to the data itself; it changes our concept of data from passive to active. Put another way,

- We stop treating each piece of data as a box with an open lid that lets us reach in and throw things around.
- We start treating each piece of data as a working machine with a closed lid and a few well-marked switches and dials.

What is described above as a "machine" may be very simple or complex on the inside; we can't tell from the outside, and we don't allow ourselves to open the machine up (except when we are absolutely sure something is wrong with its design), so we are required to do

things like flip the switches and read the dials to interact with the data. Once the machine is built, we don't want to have to think about how it operates.

You might think we are just making more work for ourselves, but this approach tends to do a nice job of preventing all kinds of things from going wrong.

It's worth noting here that the use of an OO language will not enforce proper OO design. Indeed it is possible in any language to write code that is unclear, sloppy, ill-conceived, buggy, and wobbly all over. What ruby does for you (as opposed, especially, to C++) is to make the practice of OO programming feel natural enough that even when you are working on a small scale you don't feel a necessity to resort to ugly code to save effort. We will be discussing the ways in which ruby accomplishes that admirable goal as this guide progresses; the next topic will be the "switches and dials" (object methods) and from there we'll move on to the "factories" (classes).

2.5.1.10 On Methods

What Is A Method?

What is a *method*? In OO programming, we don't think of operating on data directly from outside an object; rather, objects have some understanding of how to operate on themselves (when asked nicely to do so). You might say we pass messages to an object, and those messages will generally elicit some kind of an action or meaningful reply. This ought to happen without our necessarily knowing or caring how the object really works inside. The tasks we are allowed to ask an object to perform (or equivalently, the messages it understands) are that object's methods.

Invoking Methods Of An Object

In ruby, we invoke a method of an object with dot notation (just as in C++ or Java). The object being talked to is named to the left of the dot. '`"abcdef".length`'. Intuitively, this string object is being asked how long it is. Technically, we are invoking the `length` method of the object `abcdef`.

Other objects may have a slightly different interpretation of length, or none at all. Decisions about how to respond to a message are made on the fly, during program execution, and the action taken may change depending on what a variable refers to. What we mean by length can vary depending on what object we are talking about.

Polymorphism

An array knows something about what it means to be an array. Pieces of data in ruby carry such knowledge with them, so that the demands made on them can automatically be satisfied in the various appropriate ways. This relieves the programmer from the burden of memorizing a great many specific function names, because a relatively small number of method names, corresponding to concepts that we know how to express in natural language, can be applied to different kinds of data and the results will be what we expect. This feature of OO programming languages (which, IMHO, Java has done a poor job of exploiting) is called *polymorphism*.

Errors Are Raised

When an object receives a message that it does not understand, an error is *raised*: '`ERR: (eval):1: undefined method 'length' for 5(Fixnum)`'. So it is necessary to know what methods are acceptable to an object, though we need not know how the methods are processed.

Arguments To A Method

If arguments are given to a method, they are generally surrounded by parentheses, '`object.method(arg1, arg2)`', but they can be omitted if doing so does not cause ambiguity, '`object.method arg1, arg2`'.

The Special Variable `self`

There is a special variable self in ruby; it refers to whatever object calls a method. This happens so often that for convenience the `self.` may be omitted from method calls from an object to itself: '`self.method_name(args...)`' is the same as '`method_name(args...)`'.

What we would think of traditionally as a function call is just this abbreviated way of writing method invocations by self. This makes ruby what is called a pure object oriented language.

2.5.1.11 On Classes

In OO programming terminology, a category of objects like "dog" is called a class, and some specific object belonging to a class is called an instance of that class.

Making An Object From A Class

Generally, to make an object in ruby or any other OO language, first one defines the characteristics of a class, then creates an instance. To illustrate the process, let's first define a simple `Dog` class.

```
ruby> class Dog
    |   def speak
    |     puts "Bow Wow"
    |   end
    | end
   nil
```

In ruby, a *class definition* is a region of code between the keywords `class` and `end`. A `def` inside this region begins the definition of a method of the class, which as we discussed in the previous chapter, corresponds to some specific behavior for objects of that class.

Make A New Instance From A Class Definition

Now that we have defined a `Dog` class, we can use it to make a dog:

```
ruby> pochi = Dog.new
   #<Dog:0xbcb90>
```

We have made a new instance of the class `Dog`, and have given it the name `pochi`. The `new` method of any class makes a new instance. Because `pochi` is a `Dog` according to our class definition, it has whatever properties we decided a `Dog` should have. Since our idea of `Dog`-ness was very simple, there is just one trick we can ask `pochi` to do.

```
ruby> pochi.speak
Bow Wow
   nil
```

Making a new instance of a class is sometimes called *instantiating* that class. We need to have a dog before we can experience the pleasure of its conversation; we can't merely ask the `Dog` class to bark for us.

2.5.1.12 On Inheritance

Our classification of objects in everyday life is naturally hierarchical. We know that all cats are mammals, and all mammals are animals. Smaller classes *inherit* characteristics from the larger classes to which they belong. If all mammals breathe, then all cats breathe.

We can express this concept in ruby:

```
ruby> class Mammal
    |   def breathe
    |     puts "inhale and exhale"
    |   end
    | end
   nil
ruby> class Cat<Mammal
    |   def speak
    |     puts "Meow"
    |   end
    | end
   nil
```

Though we didn't specify how a `Cat` should breathe, every cat will inherit that behavior from the `Mammal` class since `Cat` was defined as a subclass of `Mammal`. (In OO terminology, the smaller class is a *subclass* and the larger class is a *superclass*.) Hence from a programmer's standpoint, cats get the ability to breathe for free; after we add a `speak` method, our cats can both breathe and speak.

```
ruby> tama = Cat.new
   #<Cat:0xbd80e8>
ruby> tama.breathe
inhale and exhale
   nil
ruby> tama.speak
Meow
   nil
```

Differential Programming

There will be situations where certain properties of the superclass should not be inherited by a particular subclass. Though birds generally know how to fly, penguins are a flightless subclass of birds.

```
ruby> class Bird
    |   def preen
    |     puts "I am cleaning my feathers."
```

```
    |   end
    |   def fly
    |     puts "I am flying."
    |   end
    | end
   nil
ruby> class Penguin<Bird
    |   def fly
    |     fail "Sorry. I'd rather swim."
    |   end
    | end
   nil
```

Rather than exhaustively define every characteristic of every new class, we need only to append or to redefine the differences between each subclass and its superclass. This use of inheritance is sometimes called *differential programming*. It is one of the benefits of object-oriented programming.

2.5.1.13 On Redefinition of Methods

In a subclass, we can change the behavior of the instances by redefining superclass methods.

```
ruby> class Human
    |   def identify
    |     puts "I'm a person."
    |   end
    |   def train_toll(age)
    |     if age < 12
    |       puts "Reduced fare.";
    |     else
    |       puts "Normal fare.";
    |     end
    |   end
    | end
   nil
ruby> Human.new.identify
I'm a person.
   nil
ruby> class Student1<Human
    |   def identify
    |     puts "I'm a student."
    |   end
    | end
   nil
ruby> Student1.new.identify
I'm a student.
   nil
```

Suppose we would rather enhance the superclass's `identify` method than entirely replace it. For this we can use `super`.

```
ruby> class Student2<Human
    |   def identify
    |     super
    |     puts "I'm a student too."
    |   end
    | end
   nil
ruby> Student2.new.identify
I'm a person.
I'm a student too.
   nil
```

`super` lets us pass arguments to the original method. It is sometimes said that there are two kinds of people . . .

```
ruby> class Dishonest<Human
    |   def train_toll(age)
    |     super(11) # we want a cheap fare.
    |   end
    | end
   nil
ruby> Dishonest.new.train_toll(25)
Reduced fare.
   nil

ruby> class Honest<Human
    |   def train_toll(age)
    |     super(age) # pass the argument we were given
    |   end
    | end
   nil
ruby> Honest.new.train_toll(25)
Normal fare.
   nil
```

2.5.1.14 On Access Control

Earlier, we said that ruby has no functions, only methods. However there is more than one kind of method. In this chapter we introduce *access controls*.

Consider what happens when we define a method in the "top level", not inside a class definition. We can think of such a method as analogous to a function in a more traditional language like C.

```
ruby> def square(n)
    |   n * n
    | end
   nil
ruby> square(5)
   25
```

Our new method would appear not to belong to any class, but in fact ruby gives it to the `Object` class, which is a superclass of every other class. As a result, any object should now be able to use that method. That turns out to be true, but there's a small catch: it is a *private* method of every class. We'll discuss some of what this means below, but one consequence is that it may be invoked only in function style, as here:

```
ruby> class Foo
    |   def fourth_power_of(x)
    |     square(x) * square(x)
    |   end
    | end
  nil
ruby> Foo.new.fourth_power_of 10
  10000
```

We are not allowed to explicitly apply the method to an object:

```
ruby> "fish".square(5)
ERR: (eval):1: private method ‘square’ called for "fish":String
```

This rather cleverly preserves ruby's pure-OO nature (functions are still object methods, but the receiver is self implicitly) while providing functions that can be written just as in a more traditional language.

Rationale For Private Methods: Encapsulation

A common mental discipline in OO programming, which we have hinted at in an earlier chapter, concerns the separation of specification and implementation, or what tasks an object is supposed to accomplish and how it actually accomplishes them. The internal workings of an object should be kept generally hidden from its users; they should only care about what goes in and what comes out, and trust the object to know what it is doing internally. As such it is often helpful for classes to have methods that the outside world does not see, but which are used internally (and can be improved by the programmer whenever desired, without changing the way users see objects of that class). In the trivial example below, think of engine as the invisible inner workings of the class.

```
ruby> class Test
    |   def times_two(a)
    |     puts "#{a} times two is #{engine(a)}"
    |   end
    |   def engine(b)
    |     b*2
    |   end
    |   private:engine  # this hides engine from users
    | end
   Test
ruby> test = Test.new
   #<Test:0x4017181c>
ruby> test.engine(6)
ERR: (eval):1: private method ‘engine’ called for #<Test:0x4017181c>
ruby> test.times_two(6)
6 times two is 12.
```

```
nil
```

We might have expected `test.engine(6)` to return 12, but instead we learn that `engine` is inaccessible when we are acting as a user of a `Test` object. Only other `Test` methods, such as `times_two`, are allowed to use engine. We are required to go through the public interface, which consists of the `times_two` method. The programmer who is in charge of this class can change `engine` freely (here, perhaps by changing `b*2` to `b+b`, assuming for the sake of argument that it improved performance) without affecting how the user interacts with `Test` objects. This example is of course much too simple to be useful; the benefits of access controls become more clear only when we begin to create more complicated and interesting classes.

2.5.1.15 On Singleton Methods

The behavior of an instance is determined by its class, but there may be times we know that a particular instance should have special behavior. In most languages, we must go to the trouble of defining another class, which would then only be instantiated once. In ruby we can give any object its own methods. A method given only to a single object is called a *singleton method*.

Singleton methods are often used for elements of a graphic user interface (GUI), where different actions need to be taken when different buttons are pressed.

Singleton methods are not unique to ruby, as they appear in CLOS, Dylan, etc. Also, some languages, for example, Self and NewtonScript, have singleton methods only. These are sometimes called *prototype-based* languages.

2.5.1.16 On Modules

Modules in ruby are similar to classes, except:

- A module can have no instances.
- A module can have no subclasses.
- A module is defined by `module ... end`.

Actually... the `Module` class of `module` is the superclass of the `Class` class of `class`. Got that? No? Let's move on.

Module As Collection

There are two typical uses of modules. One is to collect related methods and constants in a central location. The `Math` module in ruby's standard library plays such a role:

```
ruby> Math.sqrt(2)
   1.41421
ruby> Math::PI
   3.14159
```

The `::` operator tells the ruby interpreter which module it should consult for the value of a constant (conceivably, some module besides Math might mean something else by PI). If we want to refer to the methods or constants of a module directly without using `::`, we can `include` that module:

```
ruby> include Math
   Object
```

```
ruby> sqrt(2)
   1.41421
ruby> PI
   3.14159
```

Module As Mixin

Another use of modules is called *mixin*. Some OO programming languages, including C++, allow *multiple inheritance*, that is, inheritance from more than one superclass. A real-world example of multiple inheritance is an alarm clock; you can think of alarm clocks as belonging to the class of clocks and also the class of things with buzzers.

Ruby purposely does not implement true multiple inheritance, but the *mixin technique* is a good alternative. Remember that modules cannot be instantiated or subclassed; but if we `include` a module in a class definition, its methods are effectively appended, or *mixed in*, to the class.

Mixin As Properties

Mixin can be thought of as a way of asking for whatever particular properties we want to have. For example, if a class has a working `each` method, mixing in the standard library's `Enumerable` module gives us `sort` and `find` methods for free.

Modules Instead Of Multiple Inheritance

This use of modules gives us the basic functionality of multiple inheritance but allows us to represent class relationships with a simple tree structure, and so simplifies the language implementation considerably (a similar choice was made by the designers of Java).

2.5.1.17 On Procedure Objects (Procs)

It is often desirable to be able to specify responses to unexpected events. As it turns out, this is most easily done if we can pass blocks of code as arguments to other methods, which means we want to be able to treat code as if it were data.

A new procedure object is formed using `proc`:

```
ruby> quux = proc {
    |   puts "QUUXQUUXQUUX!!!"
    | }
   #<Proc:0x4017357c>
```

Now what `quux` refers to is an object, and like most objects, it has behavior that can be invoked. Specifically, we can ask it to execute, via its `call` method:

```
ruby> quux.call
QUUXQUUXQUUX!!!
   nil
```

So, after all that, can `quux` be used as a method argument? Sure.

```
ruby> def run( p )
    |   puts "About to call a procedure..."
    |   p.call
    |   puts "There: finished."
    | end
```

```
   nil
ruby> run quux
About to call a procedure...
QUUXQUUXQUUX!!!
There: finished.
   nil
```

The `trap` method lets us assign the response of our choice to any system signal.

```
ruby> inthandler = proc{ puts "^C was pressed." }
   #<Proc:0x401730a4>
ruby> trap "SIGINT", inthandler
   #<Proc:0x401735e0>
```

Normally pressing `^C` makes the interpreter quit. Now a message is printed and the interpreter continues running, so you don't lose the work you were doing. (You're not trapped in the interpreter forever; you can still exit by typing *`exit`*.)

Anonymous Procedure Objects

A final note before we move on to other topics: it's not strictly necessary to give a procedure object a name before binding it to a signal. An equivalent anonymous procedure object would look like '`trap "SIGINT", proc{ puts "^C was pressed." }`', or more compactly still, '` trap "SIGINT", 'puts "^C was pressed."'`'. This abbreviated form provides some convenience and readability when you write small anonymous procedures.

2.5.1.18 On Variables

Ruby has three kinds of variables, one kind of constant and exactly two pseudo-variables. The variables and the constants have no type. While untyped variables have some drawbacks, they have many more advantages and fit well with ruby's quick and easy philosophy.

No Variable Declarations

Variables must be declared in most languages in order to specify their type, modifiability (i.e., whether they are constants), and scope; since type is not an issue, and the rest is evident from the variable name as you are about to see, we do not need variable declarations in ruby.

The first character of an identifier categorizes it at a glance:

$	global variable
@	instance variable
[a-z] or _	local variable
[A-Z]	constant

Table 2.1: List of Variable Identifiers

Pseudo-Variables

The only exceptions to the above are ruby's pseudo-variables: `self`, which always refers to the currently executing object, and `nil`, which is the meaningless value assigned to uninitialized variables. Both are named as if they are local variables, but `self` is a global

variable maintained by the interpreter, and `nil` is really a constant. As these are the only two exceptions, they don't confuse things too much.

You may not assign values to `self` or `nil`. `main`, as a value of `self`, refers to the top-level object.

2.5.1.19 On Global Variables

A *global variable* has a name beginning with `$`. It can be referred to from anywhere in a program. Before initialization, a global variable has the special value `nil`.

Global variables should be used sparingly. They are dangerous because they can be written to from anywhere. Overuse of globals can make isolating bugs difficult; it also tends to indicate that the design of a program has not been carefully thought out. Whenever you do find it necessary to use a global variable, be sure to give it a descriptive name that is unlikely to be inadvertently used for something else later (calling it something like $foo as above is probably a bad idea).

Global Variables Can Be Traced

One nice feature of a global variable is that it can be traced; you can specify a procedure which is invoked whenever the value of the variable is changed.

```
ruby> trace_var :$x, proc{puts "$x is now #{$x}"}
   nil
ruby> $x = 5
$x is now 5
   5
```

When a global variable has been rigged to work as a trigger to invoke a procedure whenever changed, we sometimes call it an *active variable*. For instance, it might be useful for keeping a GUI display up to date.

List Of Major System Variables

$!	latest error message
$	location of error
$_	string last read by `gets` (has local scope)
$.	line number last read by interpreter
$&	string last matched by regexp
$~	the last regexp match, as an array of subexpressions (has local scope)
$*n*	the *nth* subexpression in the last match (same as $~[*n*])
$=	case-insensitivity flag
$/	input record separator
$\	output record separator
$0	the name of the ruby script file
$*	the command line arguments
$$	interpreter's process ID
$?	exit status of last executed child process

Table 2.2: List of Major System Variables

2.5.1.20 On Instance Variables

An instance variable has a name beginning with `@`, and its scope is confined to whatever object `self` refers to. Two different objects, even if they belong to the same class, are allowed to have different values for their instance variables. From outside the object, instance variables cannot be altered or even observed (i.e., ruby's instance variables are never `public`) except by whatever methods are explicitly provided by the programmer. As with globals, instance variables have the `nil` value until they are initialized.

Instance Variables Are Not Declared

Instance variables do not need to be declared. This indicates a flexible object structure; in fact, each instance variable is dynamically appended to an object when it is first assigned.

2.5.1.21 On Local Variables

A local variable has a name starting with a lower case letter or an underscore character (`_`). Local variables do not, like globals and instance variables, have the value `nil` before initialization.

The first assignment you make to a local variable acts something like a declaration. If you refer to an uninitialized local variable, ruby will report an error: '`ERR: (eval):1: undefined local variable or method 'foo' for main(Object)`'.

Generally, the scope of a local variable is one of:

- `proc{ ... }`
- `loop{ ... }`
- `def ... end`
- `class ... end`
- `module ... end`
- the entire script (unless one of the above applies)

`defined?` is an operator which checks whether an identifier is defined. It returns a description of the identifier if it is defined, or `nil` otherwise.

Procedure objects that live in the same scope share whatever local variables also belong to that scope. Here, the local variable `bar` is shared by `main` and the procedure objects `p1` and `p2`:

```
ruby> bar=nil
   nil
ruby> p1 = proc{|n| bar=n}
   #<Proc:0x8deb0>
ruby> p2 = proc{bar}
   #<Proc:0x8dce8>
ruby> p1.call(5)
   5
ruby> bar
   5
ruby> p2.call
   5
```

Note that the `bar=nil` at the beginning cannot be omitted; it ensures that the scope of bar will encompass `p1` and `p2`. Otherwise `p1` and `p2` would each end up with its own local variable bar, and calling `p2` would have resulted in an '`undefined local variable or method`' error. We could have said `bar=0` instead, but using `nil` is a courtesy to others who will read your code later. It indicates fairly clearly that you are only establishing scope, because the value being assigned is not intended to be meaningful.

`Proc` Objects Are Closures

A powerful feature of procedure objects follows from their ability to be passed as arguments: shared local variables remain valid even when they are passed out of the original scope.

```
ruby> def box
    |   contents = nil
    |   get = proc{contents}
    |   set = proc{|n| contents = n}
    |   return get, set
    | end
   nil
ruby> reader, writer = box
   [#<Proc:0x40170fc0>, #<Proc:0x40170fac>]
ruby> reader.call
   nil
ruby> writer.call(2)
   2
ruby> reader.call
   2
```

Ruby is particularly smart about scope. It is evident in our example that the `contents` variable is being shared between the `reader` and `writer`. But we can also manufacture multiple `reader-writer` pairs using `box` as defined above; each pair shares a `contents` variable, and the pairs do not interfere with each other.

```
ruby> reader_1, writer_1 = box
   [#<Proc:0x40172820>, #<Proc:0x4017280c>]
ruby> reader_2, writer_2 = box
   [#<Proc:0x40172668>, #<Proc:0x40172654>]
ruby> writer_1.call(99)
   99
ruby> reader_1.call
   99
ruby> reader_2.call  # nothing is in this box yet
   nil
```

This kind of programming could be considered a perverse little object-oriented framework. The `box` method acts something like a class, with `get` and `set` serving as methods (except those aren't really the method names, which could vary with each box instance) and `contents` being the lone instance variable. Of course, using ruby's legitimate class framework leads to much more readable code.

2.5.1.22 On Class Constants

A constant has a name starting with an uppercase character. It should be assigned a value at most once. In the current implementation of ruby, reassignment of a constant generates a warning but not an error (the non-ANSI version of `eval.rb` does not report the warning).

Class Constants Accessible Outside Class

Constants may be defined within classes, but unlike instance variables, they are accessible outside the class.

```
ruby> class ConstClass
    |   C1=101
    |   C2=102
    |   C3=103
    |   def show
    |     puts "#{C1} #{C2} #{C3}"
    |   end
    | end
   nil
ruby> C1
ERR: (eval):1: uninitialized constant C1
ruby> ConstClass::C1
   101
ruby> ConstClass.new.show
101 102 103
   nil
```

Constants can also be defined in modules.

```
ruby> module ConstModule
    |   C1=101
    |   C2=102
    |   C3=103
    |   def showConstants
    |     puts "#{C1} #{C2} #{C3}"
    |   end
    | end
   nil
ruby> C1
ERR: (eval):1: uninitialized constant C1
ruby> include ConstModule
   Object
ruby> C1
   101
ruby> showConstants
101 102 103
   nil
ruby> C1=99  # not really a good idea
   99
ruby> C1
```

```
   99
ruby> ConstModule::C1
   101
ruby> ConstModule::C1=99   # .. this was not allowed in earlier versions
   (eval):1: warning: already initialized constant C1
   99
ruby> ConstModule::C1  # "enough rope to shoot yourself in the foot"
   99
```

2.5.1.23 On Exception Processing and `rescue`

An executing program can run into unexpected problems. A file that it wants to read might not exist; the disk might be full when it wants to save some data; the user may provide it with some unsuitable kind of input.

A robust program will handle these situations sensibly and gracefully. Meeting that expectation can be an exasperating task. C programmers are expected to check the result of every system call that could possibly fail, and immediately decide what is to be done.

This is such a tiresome practice that programmers can tend to grow careless and neglect it, and the result is a program that doesn't handle exceptions well. On the other hand, doing the job right can make programs hard to read, because there is so much error handling cluttering up the meaningful code.

`begin` And `rescue` Blocks

In ruby, as in many modern languages, we can handle exceptions for blocks of code in a compartmentalized way, thus dealing with surprises effectively but not unduly burdening either the programmer or anyone else trying to read the code later. The block of code marked with `begin` executes until there is an exception, which causes control to be transferred to a block of error handling code, which is marked with `rescue`. If no exception occurs, the `rescue` code is not used. The following method returns the first line of a text file, or `nil` if there is an exception:

```
def first_line( filename )
  begin
    file = open("some_file")
    info = file.gets
    file.close
    info  # Last thing evaluated is the return value
  rescue
    nil   # Can't read the file? then don't return a string
  end
end
```

There will be times when we would like to be able to creatively work around a problem. Here, if the file we want is unavailable, we try to use standard input instead:

```
begin
  file = open("some_file")
rescue
  file = STDIN
```

```
end

begin
  # ... process the input ...
rescue
  # ... and deal with any other exceptions here.
end
```

`retry` can be used in the `rescue` code to start the `begin` code over again. It lets us rewrite the previous example a little more compactly:

```
fname = "some_file"
begin
  file = open(fname)
  # ... process the input ...
rescue
  fname = "STDIN"
  retry
end
```

`raise`ing Exceptions

Every ruby library raises an exception if any error occurs, and you can raise exceptions explicitly in your code too. To raise an exception, use `raise`. It takes one argument, which should be a string that describes the exception. The argument is optional but should not be omitted. It can be accessed later via the special global variable `$!`.

```
ruby> raise "test error"
   test error
ruby> begin
    |   raise "test2"
    | rescue
    |   puts "An error occurred: #{$!}"
    | end
An error occurred: test2
   nil
```

2.5.1.24 On Exception Processing And `ensure`

There may be cleanup work that is necessary when a method finishes its work. Perhaps an open file should be closed, buffered data should be flushed, etc. If there were always only one exit point for each method, we could confidently put our cleanup code in one place and know that it would be executed; however, a method might return from several places, or our intended cleanup code might be unexpectedly skipped because of an exception.

For this reason we add another keyword to the `begin...rescue...end` scheme, which is `ensure`. The `ensure` code block executes regardless of the success or failure of the `begin` block.

```
file = open("/tmp/some_file", "w")
begin
  # ... write to the file ...
```

```
rescue
  # ... handle the exceptions ...
ensure
  file.close   # ... and this always happens.
end
```

It is possible to use `ensure` without `rescue`, or vice versa, but if they are used together in the same `begin...end` block, the `rescue` must precede the `ensure`.

2.5.1.25 On Accessors

We briefly discussed instance variables in an earlier chapter, but haven't done much with them yet. An object's instance variables are its *attributes*, the things that distinguish it from other objects of the same class. It is important to be able to write and read these attributes; doing so requires methods called attribute accessors. We'll see in a moment that we don't always have to write accessor methods explicitly, but let's go through all the motions for now. The two kinds of accessors are writers and readers.

Accessors: Writers And Readers

```
ruby> class Fruit
    |   def set_kind(k)  # a writer
    |     @kind = k
    |   end
    |   def get_kind     # a reader
    |     @kind
    |   end
    | end
   nil
ruby> f1 = Fruit.new
   #<Fruit:0xfd7e7c8c>
ruby> f1.set_kind("peach")  # use the writer
   "peach"
ruby> f1.get_kind           # use the reader
   "peach"
ruby> f1                    # inspect the object
   #<Fruit:0xfd7e7c8c @kind="peach">
```

Simple enough; we can store and retrieve information about what kind of fruit we're looking at. But our method names are a little wordy. The following is more concise, and more conventional:

```
ruby> class Fruit
    |   def kind=(k)
    |     @kind = k
    |   end
    |   def kind
    |     @kind
    |   end
    | end
   nil
```

```
ruby> f2 = Fruit.new
   #<Fruit:0xfd7e7c8c>
ruby> f2.kind = "banana"
   "banana"
ruby> f2.kind
   "banana"
```

The `inspect` Method

A short digression is in order. You've noticed by now that when we try to look at an object directly, we are shown something cryptic like '`#<anObject:0x83678>`'. This is just a default behavior, and we are free to change it. All we need to do is add a method named `inspect`. It should return a string that describes the object in some sensible way, including the states of some or all of its instance variables.

```
ruby> class Fruit
    |   def inspect
    |     "a fruit of the #{@kind} variety"
    |   end
    | end
   nil
ruby> f2
   "a fruit of the banana variety"
```

`to_s` And `p` Methods

A related method is `to_s` (convert to string), which is used when printing an object. In general, you can think of `inspect` as a tool for when you are writing and debugging programs, and `to_s` as a way of refining program output. `eval.rb` (see Section D.1 "Ruby Eval Utility", page 215, uses `inspect` whenever it displays results. You can use the `p` method to easily get debugging output from programs.

```
# These two lines are equivalent:
p anObject
puts anObject.inspect
```

Making Accessors

Since many instance variables need accessor methods, Ruby provides convenient shortcuts for the standard forms.

Shortcut	**Effect**
attr_reader :v	def v; @v; end
attr_writer :v	def v=(value); @v=value; end
attr_accessor :v	attr_reader :v; attr_writer :v
attr_accessor :v, :w	attr_accessor :v; attr_accessor :w

Table 2.3: List of Accessor Shortcuts

Let's take advantage of this and add freshness information. First we ask for an automatically generated reader and writer, and then we incorporate the new information into `inspect`:

```
ruby> class Fruit
    |   attr_accessor :condition
    |   def inspect
    |     "a #{@condition} #{@kind}"
    |   end
    | end
   nil
ruby> f2.condition = "ripe"
   "ripe"
ruby> f2
   "a ripe banana"
```

If nobody eats our ripe fruit, perhaps we should let time take its toll.

```
ruby> class Fruit
    |   def time_passes
    |     @condition = "rotting"
    |   end
    | end
   nil
ruby> f2
   "a ripe banana"
ruby> f2.time_passes
   "rotting"
ruby> f2
   "a rotting banana"
```

But while playing around here, we have introduced a small problem. What happens if we try to create a third piece of fruit now? Remember that instance variables don't exist until values are assigned to them.

```
ruby> f3 = Fruit.new
ERR: failed to convert nil into String
```

It is the `inspect` method that is complaining here, and with good reason. We have asked it to report on the kind and condition of a piece of fruit, but as yet `f3` has not been assigned either attribute. If we wanted to, we could rewrite the `inspect` method so it tests instance variables using the `defined?` method and then only reports on them if they exist, but maybe that's not very useful; since every piece of fruit has a kind and condition, it seems we should make sure those always get defined somehow. That is the topic of the next chapter.

2.5.1.26 On Object Initialization

Our `Fruit` class from the previous chapter had two instance variables, one to describe the kind of fruit and another to describe its condition. It was only after writing a custom `inspect` method for the class that we realized it didn't make sense for a piece of fruit to lack those characteristics. Fortunately, ruby provides a way to ensure that instance variables always get initialized.

Default Argument Values in `initialize`

Whenever Ruby creates a new object, it looks for a method named `initialize` and executes it. So one simple thing we can do is use an `initialize` method to put default values into all the instance variables, so the `inspect` method will have something to say.

There will be times when a default value doesn't make a lot of sense. Is there such a thing as a default kind of fruit? It may be preferable to require that each piece of fruit have its kind specified at the time of its creation. To do this, we would add a formal argument to the `initialize` method. For reasons we won't get into here, arguments you supply to `new` are actually delivered to `initialize`.

```
ruby> class Fruit
    |   def initialize( k )
    |     @kind = k
    |     @condition = "ripe"
    |   end
    | end
   nil
ruby> f5 = Fruit.new "mango"
   "a ripe mango"
ruby> f6 = Fruit.new
ERR: (eval):1:in `initialize': wrong # of arguments(0 for 1)
```

Above we see that once an argument is associated with the `initialize` method, it can't be left off without generating an error. If we want to be more considerate, we can use the argument if it is given, or fall back to default values otherwise.

```
ruby> class Fruit
    |   def initialize( k="apple" )
    |     @kind = k
    |     @condition = "ripe"
    |   end
    | end
   nil
ruby> f5 = Fruit.new "mango"
   "a ripe mango"
ruby> f6 = Fruit.new
   "a ripe apple"
```

You can use *default argument values* for any method, not just `initialize`. The argument list must be arranged so that those with default values come last.

Object Reflection, Variable-Length Argument Lists, Method Overloading

Sometimes it is useful to provide several ways to initialize an object. Although it is outside the scope of this tutorial, ruby supports object reflection and variable-length argument lists, which together effectively allow method overloading.

2.5.1.27 On Nuts And Bolts

Statement Delimiters

Some languages require some kind of punctuation, often a semicolon (`;`), to end each statement in a program. Ruby instead follows the convention used in shells like `sh` and `csh`. Multiple statements on one line must be separated by semicolons, but they are not required at the end of a line; a linefeed is treated like a semicolon. If a line ends with a backslash (`\`), the linefeed following it is ignored; this allows you to have a single logical line that spans several lines.

Comments

Why write comments? Although well written code tends to be self-documenting, it is often helpful to scribble in the margins, and it can be a mistake to believe that others will be able to look at your code and immediately see it the way you do. Besides, for practical purposes, you yourself are a different person within a few days anyway; which of us hasn't gone back to fix or enhance a program after the passage of time and said, I know I wrote this, but what in blazes does it mean?

Some experienced programmers will point out, quite correctly, that contradictory or outdated comments can be worse than none at all. Certainly, comments shouldn't be a substitute for readable code; if your code is unclear, it's probably also buggy. You may find that you need to comment more while you are learning ruby, and then less as you become better at expressing your ideas in simple, elegant, readable code.

Ruby follows a common scripting convention, which is to use a pound symbol (`#`) to denote the start of a comment. Anything following an unquoted `#`, to the end of the line on which it appears, is ignored by the interpreter.

Also, to facilitate large comment blocks, the ruby interpreter also ignores anything between a line starting with `=begin` and another line starting with `=end`.

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

=begin
************************************************************************
  This is a comment block, something you write for the benefit of
  human readers (including yourself).  The interpreter ignores it.
  There is no need for a '#' at the start of every line.
************************************************************************
=end
```

Organizing Your Code

Ruby's unusually high level of dynamism means that classes, modules, and methods exist only after their defining code runs. If you're used to programming in a more static language, this can sometimes lead to surprises.

```
# The below results in an "undefined method" error:

puts successor(3)

def successor(x)
  x + 1
```

```
end
```

Although the interpreter checks over the entire script file for syntax before executing it, the `def successor ... end` code has to actually run in order to create the `successor` method. So the order in which you arrange a script can matter.

This does not, as it might seem at first glance, force you to organize your code in a strictly bottom-up fashion. When the interpreter encounters a method definition, it can safely include undefined references, as long as you can be sure they will be defined by the time the method is actually invoked:

```
# Conversion of fahrenheit to celsius, broken
# down into two steps.

def f_to_c(f)
  scale(f - 32.0)  # This is a forward reference, but it's okay.
end

def scale(x)
  x * 5.0 / 9.0
end

printf "%.1f is a comfortable temperature.\n", f_to_c(72.3)
```

So while this may seem less convenient than what you may be used to in Perl or Java, it is less restrictive than trying to write C without prototypes (which would require you to always maintain a partial ordering of what references what). Putting top-level code at the bottom of a source file always works. And even this is less of an annoyance than it might at first seem. A sensible and painless way to enforce the behavior you want is to define a `main` function at the top of the file, and call it from the bottom.

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

def main
  # Express the top level logic here...
end

# ... put support code here, organized as you see fit ...

main # ... and start execution here.
```

`load` And `require`

It also helps that ruby provides tools for breaking complicated programs into readable, reusable, logically related chunks. We have already seen the use of `include` for accessing modules (see Section 2.5.1.16 "On Modules", page 75. You will also find the `load` and `require` facilities useful.

`load` works as if the file it refers to were copied and pasted in (something like the `#include` preprocessor directive in C).

`require` is somewhat more sophisticated, causing code to be loaded at most once and only when needed.

2.5.2 Ruby Programming Wikibook

Ruby Programming Wikibook

A free online manual with beginner and intermediate content plus a thorough language reference.

2.5.3 The Pragmatic 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.

2.6 Editors and IDEs

2.7 Further Reading

Programming Ruby
: by Marek Hulan, et al., contains succinct advanced information about Ruby. See Section C.2 "Programming Ruby by Hulan", page 188. (See item *ProgrammingRuby* in "Bibliography", page 239.)

Pragmatic Programmer's Guide — The Pickaxe (aka Programming Ruby)
: by Dave Thomas and Andy Hunt. The place where just about every Ruby beginner starts. See Section 2.5.3 "The Pragmatic Programmer's Guide", page 90.

The Ruby Way
: by Hal Fulton. When you're reasonably familiar with Ruby, but want to learn more and see how to accomplish all sorts of tasks with it, then `http://therubyway.io`, is the logical next step.

The Bastards Book Of Ruby
: A Programming Primer for Counting and Other Unconventional Tasks by Dan Nguyen `http://ruby.bastardsbook.com/`

The Bastards Book of Ruby is an introduction to programming and its practical uses for journalists, researchers, scientists, analysts, and anyone else whose job is to seek out, make sense from, and show the hard-to-find data.

The Bastards Book Of Regular Expressions

A free guide to finding patterns in text by Dan Nguyen. `http://regex.bastardsbook.com/` and `http://regex.bastardsbook.com/files/bastards-regexes.pdf`

3 Implementations

3.1 YARV Implementation

YARV stands for "Yet Another Ruby VM" and is a bytecode interpreter that was developed for the Ruby programming language by Koichi Sasada. See YARV. The goal of the project was to greatly reduce the execution time of Ruby programs. Since YARV has become the official Ruby interpreter for Ruby 1.9, it is also named KRI ("Koichi's Ruby Interpreter"), in the same vein as the original Ruby MRI, named for Ruby's creator Yukihiro Matsumoto.

YARV was merged into the Ruby Subversion repository on January 1, 2007. It was released as part of Ruby 1.9.0 on December 26, 2007, replacing Ruby MRI.

3.2 JRuby Implementation

The Ruby Programming Language On The JVM

JRuby 9.2.x is our new major version of JRuby. It is expected to be compatible with Ruby 2.5.x and stay in sync with C Ruby. JRuby 9.2.0.0 is our first release for 2.5 support.

JRuby

Latest Release

JRuby Wiki

3.3 Rubinius Implementation

Rubinius is an alternative Ruby implementation created by Evan Phoenix.

Rubinius follows in the Lisp and Smalltalk traditions, by natively implementing as much of Ruby as possible in Ruby code.

The current `Rubinius` release is version 3.107. `Rubinius` provides Homebrew binaries that should be compatible with 10.8 (Mountain Lion) and newer OS X releases. To install `Rubinius` on Homebrew, follow these steps:

```
$ brew update
$ brew tap rubinius/apps
$ brew install rubinius
```

3.4 The Rubinius Book

The Rubinius Book

4 My Ruby Reference

4.1 Data Types

4.1.1 Number Data Types

4.1.2 Character Data Type

4.1.3 String Data Type

4.1.4 Array Data Type

All arrays are instances of the class `Array`. They may contain data of any type, including mixed types. An array constant is delimited by brackets (`[...]`). Arrays are zero-based and dynamic.

An array variable uses brackets enclosing an expression to index into its array object. This allows the variable to both reveal its content and update its content.

```
[1, 2, 3]
["a", "b", "c"]
[1, "a", 2, "b", 3, "c"]

arr = [1, 2, 3]
arr[0]     # ⇒ 1
arr[3] = 4 # ⇒ [1, 2, 3, 4]
```

Special Forms for String Arrays

Because the array of strings is so common (and inconvenient to type), a special syntax has been created for it: no quotes or commas are needed. Whitespace separates the elements (so strings with whitespace will need quotes).

```
%w[alpha beta gamma]
%w(alpha beta gamma)
%w/alpha beta gamma/
```

4.1.5 Hash Data Type

A *hash* is a set of associations between paired pieces of data. Each hash is an instance of the class `Hash`.

A hash constant is represented between delimiting braces with the fat arrow separating the individual keys and values: '`{key => value}`'.

Hashes also have an additional syntax that creates keys that are instances of the `Symbol` class: '`{symbol: value}`'.

Hash variable content is accessed using brackets enclosing the key: '`hsh["key"]`' or symbol: '`hsh[:symbol]`'.

```
{odds => [1, 3, 5, 7, 9], evens => [2, 4, 6, 8]

{hydrogen: 1, helium: 2, carbon: 3}
```

4.2 Operators

Most operators are actually method calls. For example, '`a + b`' is interpreted as '`a.+(b)`', where the `+` method in the object referred to by variable `a` is called with `b` as its argument.

For each operator ('`+ - * / % ** & | ^ << >> && ||`'), there is a corresponding form of abbreviated assignment operator ('`+= -=`' etc.).

Operators that are methods may be overridden.

4.2.1 Arithmetic Operators

`+`	Addition — Adds values on either side of the operator.
`-`	Subtraction — Subtracts right hand operand from left hand operand.
`*`	Multiplication — Multiplies values on either side of the operator.
`/`	Division — Divides left hand operand by right hand operand.
`%`	Modulus — Divides left hand operand by right hand operand and returns remainder.
`**`	Exponent Performs exponential (power) calculation on operators.

Table 4.1: Table of Arithmetic Operators

4.2.2 Comparison Operators

4.2.3 Assighnment Operators

4.2.4 Bitwise Operators

4.2.5 Logical Operators

4.2.6 Ternary Operator

4.2.7 Range Operators

4.2.8 Defined? Operators

4.2.9 Dot and Colon Operators

4.2.10 Operator Precedence

4.3 Keywords

- BEGIN
- END

- alias
- and
- begin
- break
- case
- class
- def
- defined?
- do
- else
- elsif
- end
- ensure
- false
- for
- if
- in
- module
- next
- nil
- not
- or
- redo
- rescue
- retry
- return
- self
- super
- then
- true
- undef
- unless
- unless
- until
- when
- while
- yield

4.4 Variables

Variables do not have types, but the objects they refer to do have types.

Local	Begin with a lowercase letter or underscore
Global	Begin with a *$*; special variables starting with a *$* are set by the Ruby interpreter directly
Instance	Begin with an `@`
Class	Begin with two `@@`
Constant	Begin with a capital letter

4.5 Comments

- `#` to *eol*
- `=begin ... =end`
- Comments immediately before definitions typically document the thing that is about to be defined. The embedded documentation can often be retrieved from the program text with an external tool. Typical documentation can run to several comment-lines in a row.

```
# Some documentation
# Some more documentation
class Clazz
  some content ...
end
```

4.6 Equality

Ruby objects implement five different methods that test for equality.

`equal?` — Object Identity

The most basic comparison is the `equal?` method (which comes from `BasicObject`. It returns `true` if its receiver and parameter have the same object ID. This is a fundamental part of the semantics of objects, and shouldn't be overridden.

`==` — Value Equality

The most common test for equality uses the method `==`, which tests the values of its receiver with its argument.

`eql?` — Strict Equality

Next on the scale of abstraction is the method `eql?`, which is part of `Object`. (Actually, `eql?` is implemented in the `Kernel` module, which is mixed into `Object`. Like the `==` operator, `eql?` compares its receiver and its argument, but it is slightly stricter. For example, different numeric objects will be coerced into a common type when compared using `==`, but objects of different types will never test equal using `eql?`.

```
1 == 1.0   # ⇒ true
```

```
1.eql? 1.0 # ⇒ false
```

The `eql?` method exists for one reason: It is used to compare the values of hash keys. If you want to override Ruby's default behavior when using your objects as hash keys, you'll need to override the methods `eql?` and `hash` for those objects.

`===` — Triple Equal Case Equality

The `===` method is used to compare the target in a `case` statement against each of the selectors, using `selector === target`. This rule allows Ruby `case` statements to be intuitive in practice. For example, you can `switch` on the class of an object:

```
case an_object
  when String
    puts "It's a string."
  when Numeric
    puts "It's a number."
  else
    puts "It's something else entirely."
  end
```

This works because class `Module` implements `===` to test whether its parameter is an instance of its receiver (or the receiver's parents). Therefore, if `an_object` is the string `cat`, the expression `String === an_object` would be true, and the first clause in the `case` statement would fire.

`=~` — Match Operator

This is used by strings and regular expressions to implement pattern matching. It may freely be overloaded.

Negated Equals `!=` and `!~`

The equality tests `==` and `=~` also have the negated forms `!=` and `!~`, respectively. These are implemented internally by reversing the sense of the non-negated forms. This means that if you implement a method, you get the negated form for free.

4.7 Dynamic Features

4.7.1 Dynamic Instance Variables

Is An Instance Variable Defined?

There is a special method to determine whether an instance variable is already defined: `instance_variable_defined?(<sym>)`.

Get And Set Instance Variables Using A String

Ruby has methods that can retrieve or assign instance variables given the variable name as a string:

- `instance_variable_set(<string>)`

- `instance_variable_get(<string>`

```
obj.instance_variable_set("@alpha", 234)
obj.instance_variable_get("@alpha")
```

You must use the `@` sign.

4.7.2 Dynamic Objects

Sending A Message To An Object

Every time you invoke a method, you are sending a message to an object. Most of the time, these messages are hard-coded as in a static language, but they need not always be. We can write code that determines at runtime which method to call. The `send()` method will allow us to us a `Symbol` to represent a method name. `__SEND__` is an alias. `public_send()` is used to send a message to only public methods.

4.7.3 Eval'ing Code

The global function `eval()` compiles and executes a string that contains a fragment of Ruby code. This is a powerful (but extremely dangerous) mechanism, because it allows you to build up code to be executed at runtime.

```
eval("2 + 3") # ⇒ 5
```

Other Methods That Evaluate Code Dynamically

1. class_eval
2. module_eval
3. instance_eval

The first two are synonyms. All three do effectively the same thing: evaluate a string or a block, but while doing so they change the value of `self` to their own receiver. The most common use of `class_eval` allows you to add methods to a class when all you have is a reference to that class.

You can see an example in the Standard Library `delegate.rb`.

Evaluating Local Variables In New Scope

The `eval()` method also makes it possible to evaluate local variables in a context outside their scope. Ruby associates local variables with blocks, with high-level definition constructs (class, module, and method definitions), and with the top-level of your program (the code outside any definition constructs). Associated with each of these scopes is the binding of variables, along with housekeeping details.

You can encapsulate the current binding in an object using the method `Kernel#binding`. Having done that, you can pass the binding as the second parameter to `eval()`, setting the execution context for the code being evaluated.

```
def some_method
  a = "local variable"
  return binding
end
```

```
the_binding = some_method
eval "a", the_binding # ⇒ "local variable"
```

The presence of a block associated with a method is stored as part of the binding, enabling:

```
def some_method
  return binding
end

the_binding = some_method { puts "hello" }
eval "yield", the_binding # ⇒ "hello"
```

4.7.4 Retrieving By Name

Retrieving A Constant From A Module Or Class

The `const_get` method retrieves the value of a constant (by name) from the module or class to which it belongs.

```
str = "PI"
Math.const_get(str) # ⇒ Math::PI
```

This is a way of avoiding the use of `eval`, both dangerous and inelegant. This type of solution is better code, its computationally cheaper, and it's safer. `const_get` is much faster than `eval`.

The usefulness of `const_get` is that it is easier to read, more specific, and more self-documenting. This is the real reason to use it.

Other similar methods are:

- `instance_variable_set`
- `instance_variable_get`
- `define_method`

Retrieving A Class By Name From A String

Given a string containing the name of a class, how to create an instance of that class? Classes in Ruby are normally named as constants in the "global" namespace — that is, members of `Object`. That means the proper way is with `const_get`:

```
classname = "Array"
klass = Object.const_get(classname)
x = klass.new(4, 1) # ⇒ [1, 1, 1, 1]
```

If the constant is inside a namespace, just provide a string with namespaces delimited by two colons (as if you were writing Ruby directly):

```
class Alpha
  class Beta
    class Gamma
      FOOBAR = 237
    end
```

```
  end
end

str = "Alpha::Beta::Gamma::FOOBAR"
val = Object.const_get(str) # ⇒ 237
```

4.7.5 Defining Methods Dynamically

`define_method` Definition

`define_method` (*`symbol`*, *`method`*) ↦ *Symbol* [Method on `Module`]
`define_method` (*`symbol`*) { *`block`* } ↦ *Symbol* [Method on `Module`]

Defines an instance method in the receiver. The method parameter can be a Proc, a Method or an UnboundMethod object. If a block is specified, it is used as the method body. This block is evaluated using `instance_eval`, a point that is tricky to demonstrate because `define_method` is private. (This is why we resort to the `send` hack in this example.)

`define_method` Example

```
class A
  def fred
    puts "In Fred"
  end
  def create_method(name, &block)
    self.class.send(:define_method, name, &block)
  end
  define_method(:wilma) { puts "Charge it!" }
end

class B < A
  define_method(:barney, instance_method(:fred))
end

a = B.new
a.barney # ⊣ In Fred
a.wilma # ⊣ Charge it!
a.create_method(:betty) { p self }
a.betty # ⊣ #<B:0x401b39e8>
```

`define_method` Description

`define_method` allows you to add a method to a class or object at runtime. `define_method` takes a symbol (for the name of the method) and a block (for the body of the method):

```
if today =~ /Saturday|Sunday/
  define_method(:activity) {puts "Playing"}
else
  define_method(:activity) {puts "Working"}
```

```
end
```

However, `define_method` is private. This means that calling it from inside a class definition or method will work.

```
class MyClass
  define_method(:body_method) { puts "The class body" }

  def self.new_method(name, &block)
    define_method(name, &block)
  end
end

MyClass.new_method(:class_method) { puts "A class method" }

x = MyClass.new
x.body_method # ⊣ "The class body"
x.class_method  # ⊣ "A class method"
```

We can even create an instance method that dynamically defines other instance methods:

```
class MyClass
  def new_method(name, &block)
    self.class.send(:define_method, name, &block)
  end
end

x = MyClass.new
x.new_method(:mymeth) { puts "An instance method" }
x.mymeth # ⊣ "An instance method"
```

We are still defining an instance method dynamically; but the means of invoking `new_method` has changed. The `send` method is used to circumvent the privacy of `define_method`. This works because `send` always allows you to call private methods. (This "loophole", as some would call it, has to be used responsibly.)

`define_method` takes a block, and a block is a closure. This means that, unlike an ordinary method definition, we are capturing context when we define the method. The following example illustrates the point:

```
class MyClass
  def self.new_method(name, &block)
    define_method(name, &block)
  end
end

a, b = 3, 79
MyClass.new_method(:compute) { a * b }

x = MyClass.new
puts x.compute   # ⊣ 237
```

```
a, b = 23, 34
puts x.compute   # ⊣ 552
```

Appendix A Ruby-Doc

Help and documentation for the Ruby programming language.

- Ruby-Doc Core Reference Home
- Core API

  These are the API documents for the base classes and modules in the current stable release of Ruby 2.5.
- Standard Library API

  These are the API documents for the standard library classes and modules in version 2.5
- Getting Started

  A collection of resources for those just starting out with Ruby.
- Ruby-Doc Downloads
- The Ruby Specification Project

A.1 API Documentation

This is the API documentation for Ruby 2.5.1.

Section A.1.1 “Files API”, page 105,
Section A.1.2 “Classes And Modules API”, page 106,
Section A.1.3 “Methods API”, page 112,

A.1.1 Files API

Grammar http://ruby-doc.org/core-2.5.1/_lib/racc/rdoc/grammar_en_rdoc.html

Contributing
http://ruby-doc.org/core-2.5.1/doc/contributing_rdoc.html

DTrace Probes
http://ruby-doc.org/core-2.5.1/doc/dtrace_probes_rdoc.html

Extension http://ruby-doc.org/core-2.5.1/doc/extension_rdoc.html

Globals http://ruby-doc.org/core-2.5.1/doc/globals_rdoc.html

Keywords http://ruby-doc.org/core-2.5.1/doc/keywords_rdoc.html

Marshall http://ruby-doc.org/core-2.5.1/doc/marshal_rdoc.html

RegExp http://ruby-doc.org/core-2.5.1/doc/regexp_rdoc.html

Security http://ruby-doc.org/core-2.5.1/doc/security_rdoc.html

Standard Library
http://ruby-doc.org/core-2.5.1/doc/standard_library_rdoc.html

Syntax http://ruby-doc.org/core-2.5.1/doc/syntax_rdoc.html

Assignment
http://ruby-doc.org/core-2.5.1/doc/syntax/assignment_rdoc.html

Calling Methods
http://ruby-doc.org/core-2.5.1/doc/syntax/calling_methods_rdoc.html

Control Expressions
http://ruby-doc.org/core-2.5.1/doc/syntax/control_expressions_rdoc.html

Exceptions
http://ruby-doc.org/core-2.5.1/doc/syntax/exceptions_rdoc.html

Literals http://ruby-doc.org/core-2.5.1/doc/syntax/literals_rdoc.html

Methods http://ruby-doc.org/core-2.5.1/doc/syntax/methods_rdoc.html

Miscellaneous
http://ruby-doc.org/core-2.5.1/doc/syntax/miscellaneous_rdoc.html

Modules and Classes
http://ruby-doc.org/core-2.5.1/doc/syntax/modules_and_classes_rdoc.html

Precedence
http://ruby-doc.org/core-2.5.1/doc/syntax/precedence_rdoc.html

Refinements
http://ruby-doc.org/core-2.5.1/doc/syntax/refinements_rdoc.html

README http://ruby-doc.org/core-2.5.1/sample/drb/README_rdoc.html

A.1.2 Classes And Modules API

`ARGF` Class

`ArgumentError`
Class

`Array` Class

`BasicObject`
Class

`Binding` Class

`Class` Class

`ClosedQueueError`
Class

`Comparable`
Module

`Complex` Class

`Complex::compatible`
Class

`ConditionVariable`
Class

`Continuation`
Class

`Data` Class

`Dir` Class

`ENV` Class

`EOFError` Class

`Encoding` Class

`Encoding::CompatibilityError`
Class

`Encoding::Converter`
Class

`Encoding::ConverterNotFoundError`
Class

`Encoding::InvalidByteSequenceError`
Class

`Encoding::UndefinedConversionError`
Class

`EncodingError`
Class

`Enumerable`
: Module

`Enumerator`
: Class

`Enumerator::Generator`
: Class

`Enumerator::Lazy`
: Class

`Enumerator::Yielder`
: Class

`Errno`
: Module

`Exception`
: Class

`FalseClass`
: Class

`Fiber`
: Class

`FiberError`
: Class

`File`
: Class

`File::Constants`
: Module

`File::Stat`
: Class

`FileTest`
: Module

`Float`
: Class

`FloatDomainError`
: Class

`FrozenError`
: Class

`GC`
: Module

`GC::Profiler`
: Module

`Hash`
: Class

`IO`
: Class

`IO::EAGAINWaitReadable`
: Class

`IO::EAGAINWaitWritable`
: Class

`IO::EINPROGRESSWaitReadable`
Class

`IO::EINPROGRESSWaitWritable`
Class

`IO::EWOULDBLOCKWaitReadable`
Class

`IO::EWOULDBLOCKWaitWritable`
Class

`IO::WaitReadable`
Module

`IO::WaitWritable`
Module

`IOError` Class

`IndexError`
Class

`Integer` Class

`Interrupt`
Class

`Kernel` Module

`KeyError` Class

`LoadError`
Class

`LocalJumpError`
Class

`Marshal` Module

`MatchData`
Class

`Math` Module

`Math::DomainError`
Class

`Method` Class

`Module` Class

`Mutex` Class

`NameError`
Class

`NilClass` Class

`NoMemoryError`
: Class

`NoMethodError`
: Class

`NotImplementedError`
: Class

`Numeric` Class

`Object` Class

`ObjectSpace`
: Module

`ObjectSpace::WeakMap`
: Class

`Proc` Class

`Process` Module

`Process::GID`
: Module

`Process::Status`
: Class

`Process::Sys`
: Module

`Process::UID`
: Module

`Process::Waiter`
: Class

`Queue` Class

`Random` Class

`Random::Formatter`
: Module

`Range` Class

`RangeError`
: Class

`Rational` Class

`Rational::compatible`
: Class

`Regexp` Class

`RegexpError`
: Class

`RubyVM` Class

`RubyVM::InstructionSequence`
Class

`RuntimeError`
Class

`ScriptError`
Class

`SecurityError`
Class

`Signal` Module

`SignalException`
Class

`SizedQueue`
Class

`StandardError`
Class

`StopIteration`
Class

`String` Class

`Struct` Class

`Symbol` Class

`SyntaxError`
Class

`SystemCallError`
Class

`SystemExit`
Class

`SystemStackError`
Class

`Thread` Class

`Thread::Backtrace`
Class

`Thread::Backtrace::Location`
Class

`ThreadError`
Class

`ThreadGroup`
Class

`Time` Class

`TracePoint`
Class

`TrueClass`
Class

`TypeError`
Class

`UnboundMethod`
Class

`UncaughtThrowError`
Class

`UnicodeNormalize`
Module

`Warning` Module

`Warning::buffer`
Class

`ZeroDivisionError`
Class

`fatal` Class

A.1.3 Methods API

`===` `SystemCallError::===` (Class Method)

`DEBUG` `Thread::DEBUG` (Class Method)

`DEBUG=` `Thread::DEBUG=` (Class Method)

`[]` `Array::[]` (Class Method)

`[]` `Dir::[]` (Class Method)

`[]` `ENV::[]` (Class Method)

`[]` `Hash::[]` (Class Method)

`[]=` `ENV::[]=` (Class Method)

`_id2ref` `ObjectSpace::_id2ref` (Class Method)

`abort` `Process::abort` (Class Method)

`abort_on_exception`
`Thread::abort_on_exception` (Class Method)

`abort_on_exception=`
`Thread::abort_on_exception=` (Class Method)

`absolute_path`
`File::absolute_path` (Class Method)

`acos` `Math::acos` (Class Method)

`acosh` `Math::acosh` (Class Method)

`add_stress_to_class`
`GC::add_stress_to_class` (Class Method)

`aliases` `Encoding::aliases` (Class Method)

`all_symbols`
`Symbol::all_symbols` (Class Method)

`argv0` `Process::argv0` (Class Method)

`asciicompat_encoding`
`Encoding/Converter::asciicompat_encoding` (Class Method)

`asin` `Math::asin` (Class Method)

`asinh` `Math::asinh` (Class Method)

`assoc` `ENV::assoc` (Class Method)

`at` `Time::at` (Class Method)

`atan` `Math::atan` (Class Method)

`atan2` `Math::atan2` (Class Method)

`atanh` `Math::atanh` (Class Method)

atime `File::atime` (Class Method)

basename `File::basename` (Class Method)

binread `IO::binread` (Class Method)

binwrite `IO::binwrite` (Class Method)

birthtime
`File::birthtime` (Class Method)

blockdev?
`File::blockdev?` (Class Method)

cbrt `Math::cbrt` (Class Method)

change_privilege
`Process/GID::change_privilege` (Class Method)

change_privilege
`Process/UID::change_privilege` (Class Method)

chardev? `File::chardev?` (Class Method)

chdir `Dir::chdir` (Class Method)

children `Dir::children` (Class Method)

chmod `File::chmod` (Class Method)

chown `File::chown` (Class Method)

chroot `Dir::chroot` (Class Method)

clear `ENV::clear` (Class Method)

clear `GC/Profiler::clear` (Class Method)

clock_getres
`Process::clock_getres` (Class Method)

clock_gettime
`Process::clock_gettime` (Class Method)

compatible?
`Encoding::compatible?` (Class Method)

compile `Regexp::compile` (Class Method)

compile `RubyVM/InstructionSequence::compile` (Class Method)

compile_file
`RubyVM/InstructionSequence::compile_file` (Class Method)

compile_option
`RubyVM/InstructionSequence::compile_option` (Class Method)

compile_option=
`RubyVM/InstructionSequence::compile_option=` (Class Method)

disassemble
: `RubyVM/InstructionSequence::disassemble` (Class Method)

dump
: `Marshal::dump` (Class Method)

each
: `ENV::each` (Class Method)

each_child
: `Dir::each_child` (Class Method)

each_key
: `ENV::each_key` (Class Method)

each_object
: `ObjectSpace::each_object` (Class Method)

each_pair
: `ENV::each_pair` (Class Method)

each_value
: `ENV::each_value` (Class Method)

egid
: `Process::egid` (Class Method)

egid=
: `Process::egid=` (Class Method)

eid
: `Process/GID::eid` (Class Method)

eid
: `Process/UID::eid` (Class Method)

empty?
: `Dir::empty?` (Class Method)

empty?
: `ENV::empty?` (Class Method)

empty?
: `File::empty?` (Class Method)

enable
: `GC::enable` (Class Method)

enable
: `GC/Profiler::enable` (Class Method)

enabled?
: `GC/Profiler::enabled?` (Class Method)

entries
: `Dir::entries` (Class Method)

erf
: `Math::erf` (Class Method)

erfc
: `Math::erfc` (Class Method)

escape
: `Regexp::escape` (Class Method)

euid
: `Process::euid` (Class Method)

euid=
: `Process::euid=` (Class Method)

exception
: `Exception::exception` (Class Method)

exclusive
: `Thread::exclusive` (Class Method)

exec
: `Process::exec` (Class Method)

executable?
: `File::executable?` (Class Method)

executable_real?
: `File::executable_real?` (Class Method)

exist?
: `Dir::exist?` (Class Method)

exist?
: `File::exist?` (Class Method)

exists?
: `Dir::exists?` (Class Method)

exists?
: `File::exists?` (Class Method)

exit
: `Process::exit` (Class Method)

exit
: `Thread::exit` (Class Method)

exit!
: `Process::exit!` (Class Method)

exp
: `Math::exp` (Class Method)

expand_path
: `File::expand_path` (Class Method)

extname
: `File::extname` (Class Method)

fetch
: `ENV::fetch` (Class Method)

file?
: `File::file?` (Class Method)

find
: `Encoding::find` (Class Method)

fnmatch
: `File::fnmatch` (Class Method)

fnmatch?
: `File::fnmatch?` (Class Method)

for_fd
: `IO::for_fd` (Class Method)

foreach
: `Dir::foreach` (Class Method)

foreach
: `IO::foreach` (Class Method)

fork
: `Process::fork` (Class Method)

fork
: `Thread::fork` (Class Method)

frexp
: `Math::frexp` (Class Method)

from_name
: `Process/GID::from_name` (Class Method)

from_name
: `Process/UID::from_name` (Class Method)

ftype
: `File::ftype` (Class Method)

gamma
: `Math::gamma` (Class Method)

garbage_collect
: `ObjectSpace::garbage_collect` (Class Method)

getegid
: `Process/Sys::getegid` (Class Method)

geteuid
: `Process/Sys::geteuid` (Class Method)

`getgid` `Process/Sys::getgid` (Class Method)

`getpgid` `Process::getpgid` (Class Method)

`getpgrp` `Process::getpgrp` (Class Method)

`getpriority`
`Process::getpriority` (Class Method)

`getrlimit`
`Process::getrlimit` (Class Method)

`getsid` `Process::getsid` (Class Method)

`getuid` `Process/Sys::getuid` (Class Method)

`getwd` `Dir::getwd` (Class Method)

`gid` `Process::gid` (Class Method)

`gid=` `Process::gid=` (Class Method)

`glob` `Dir::glob` (Class Method)

`gm` `Time::gm` (Class Method)

`grant_privilege`
`Process/GID::grant_privilege` (Class Method)

`grant_privilege`
`Process/UID::grant_privilege` (Class Method)

`groups` `Process::groups` (Class Method)

`groups=` `Process::groups=` (Class Method)

`grpowned?`
`File::grpowned?` (Class Method)

`handle_interrupt`
`Thread::handle_interrupt` (Class Method)

`has_key?` `ENV::has_key?` (Class Method)

`has_value?`
`ENV::has_value?` (Class Method)

`home` `Dir::home` (Class Method)

`hypot` `Math::hypot` (Class Method)

`identical?`
`File::identical?` (Class Method)

`include?` `ENV::include?` (Class Method)

`index` `ENV::index` (Class Method)

`initgroups`
`Process::initgroups` (Class Method)

log Math::log (Class Method)

log10 Math::log10 (Class Method)

log2 Math::log2 (Class Method)

lstat File::lstat (Class Method)

lutime File::lutime (Class Method)

main Thread::main (Class Method)

malloc_allocated_size
GC::malloc_allocated_size (Class Method)

malloc_allocations
GC::malloc_allocations (Class Method)

maxgroups
Process::maxgroups (Class Method)

maxgroups=
Process::maxgroups= (Class Method)

member? ENV::member? (Class Method)

mkdir Dir::mkdir (Class Method)

mkfifo File::mkfifo (Class Method)

mktime Time::mktime (Class Method)

mtime File::mtime (Class Method)

name_list
Encoding::name_list (Class Method)

nesting Module::nesting (Class Method)

new Array::new (Class Method)

new BasicObject::new (Class Method)

new Class::new (Class Method)

new ConditionVariable::new (Class Method)

new Dir::new (Class Method)

new Encoding/Converter::new (Class Method)

new Enumerator::new (Class Method)

new Enumerator/Lazy::new (Class Method)

new Exception::new (Class Method)

new File::new (Class Method)

new File/Stat::new (Class Method)

new Hash::new (Class Method)

new IO::new (Class Method)
new Module::new (Class Method)
new Mutex::new (Class Method)
new NameError::new (Class Method)
new NoMethodError::new (Class Method)
new Proc::new (Class Method)
new Queue::new (Class Method)
new Random::new (Class Method)
new Range::new (Class Method)
new Regexp::new (Class Method)
new RubyVM/InstructionSequence::new (Class Method)
new SignalException::new (Class Method)
new SizedQueue::new (Class Method)
new String::new (Class Method)
new Struct::new (Class Method)
new SyntaxError::new (Class Method)
new SystemCallError::new (Class Method)
new SystemExit::new (Class Method)
new Thread::new (Class Method)
new Time::new (Class Method)
new TracePoint::new (Class Method)
new UncaughtThrowError::new (Class Method)
new_seed Random::new_seed (Class Method)
now Time::now (Class Method)
of RubyVM/InstructionSequence::of (Class Method)
open Dir::open (Class Method)
open File::open (Class Method)
open IO::open (Class Method)
owned? File::owned? (Class Method)
pass Thread::pass (Class Method)
path File::path (Class Method)
pending_interrupt?
Thread::pending_interrupt? (Class Method)

pid `Process::pid` (Class Method)

pipe `IO::pipe` (Class Method)

pipe? `File::pipe?` (Class Method)

polar `Complex::polar` (Class Method)

popen `IO::popen` (Class Method)

ppid `Process::ppid` (Class Method)

pwd `Dir::pwd` (Class Method)

quote `Regexp::quote` (Class Method)

rand `Random::rand` (Class Method)

rassoc `ENV::rassoc` (Class Method)

raw_data `GC/Profiler::raw_data` (Class Method)

re_exchange
`Process/GID::re_exchange` (Class Method)

re_exchange
`Process/UID::re_exchange` (Class Method)

re_exchangeable?
`Process/GID::re_exchangeable?` (Class Method)

re_exchangeable?
`Process/UID::re_exchangeable?` (Class Method)

read `IO::read` (Class Method)

readable?
`File::readable?` (Class Method)

readable_real?
`File::readable_real?` (Class Method)

readlines
`IO::readlines` (Class Method)

readlink `File::readlink` (Class Method)

realdirpath
`File::realdirpath` (Class Method)

realpath `File::realpath` (Class Method)

rect `Complex::rect` (Class Method)

rectangular
`Complex::rectangular` (Class Method)

rehash `ENV::rehash` (Class Method)

reject `ENV::reject` (Class Method)

reject! ENV::reject! (Class Method)

remove_stress_to_class
GC::remove_stress_to_class (Class Method)

rename File::rename (Class Method)

replace ENV::replace (Class Method)

report GC/Profiler::report (Class Method)

report_on_exception
Thread::report_on_exception (Class Method)

report_on_exception=
Thread::report_on_exception= (Class Method)

restore Marshal::restore (Class Method)

result GC/Profiler::result (Class Method)

rid Process/GID::rid (Class Method)

rid Process/UID::rid (Class Method)

rmdir Dir::rmdir (Class Method)

search_convpath
Encoding/Converter::search_convpath (Class Method)

select ENV::select (Class Method)

select IO::select (Class Method)

select! ENV::select! (Class Method)

setegid Process/Sys::setegid (Class Method)

seteuid Process/Sys::seteuid (Class Method)

setgid Process/Sys::setgid (Class Method)

setgid? File::setgid? (Class Method)

setpgid Process::setpgid (Class Method)

setpgrp Process::setpgrp (Class Method)

setpriority
Process::setpriority (Class Method)

setproctitle
Process::setproctitle (Class Method)

setregid Process/Sys::setregid (Class Method)

setresgid
Process/Sys::setresgid (Class Method)

setresuid
Process/Sys::setresuid (Class Method)

setreuid `Process/Sys::setreuid` (Class Method)

setrgid `Process/Sys::setrgid` (Class Method)

setrlimit
`Process::setrlimit` (Class Method)

setruid `Process/Sys::setruid` (Class Method)

setsid `Process::setsid` (Class Method)

setuid `Process/Sys::setuid` (Class Method)

setuid? `File::setuid?` (Class Method)

shift `ENV::shift` (Class Method)

sid_available?
`Process/GID::sid_available?` (Class Method)

sid_available?
`Process/UID::sid_available?` (Class Method)

signame `Signal::signame` (Class Method)

sin `Math::sin` (Class Method)

sinh `Math::sinh` (Class Method)

size `ENV::size` (Class Method)

size `File::size` (Class Method)

size? `File::size?` (Class Method)

socket? `File::socket?` (Class Method)

spawn `Process::spawn` (Class Method)

split `File::split` (Class Method)

sqrt `Integer::sqrt` (Class Method)

sqrt `Math::sqrt` (Class Method)

srand `Random::srand` (Class Method)

start `GC::start` (Class Method)

start `Thread::start` (Class Method)

stat `File::stat` (Class Method)

stat `GC::stat` (Class Method)

stat `RubyVM::stat` (Class Method)

stat `TracePoint::stat` (Class Method)

sticky? `File::sticky?` (Class Method)

stop `Thread::stop` (Class Method)

store `ENV::store` (Class Method)

`union` `Regexp::union` (Class Method)

`unlink` `Dir::unlink` (Class Method)

`unlink` `File::unlink` (Class Method)

`update` `ENV::update` (Class Method)

`urandom` `Random::urandom` (Class Method)

`used_modules`
`Module::used_modules` (Class Method)

`utc` `Time::utc` (Class Method)

`utime` `File::utime` (Class Method)

`value?` `ENV::value?` (Class Method)

`values` `ENV::values` (Class Method)

`values_at`
`ENV::values_at` (Class Method)

`verify_internal_consistency`
`GC::verify_internal_consistency` (Class Method)

`wait` `Process::wait` (Class Method)

`wait2` `Process::wait2` (Class Method)

`waitall` `Process::waitall` (Class Method)

`waitpid` `Process::waitpid` (Class Method)

`waitpid2` `Process::waitpid2` (Class Method)

`world_readable?`
`File::world_readable?` (Class Method)

`world_writable?`
`File::world_writable?` (Class Method)

`writable?`
`File::writable?` (Class Method)

`writable_real?`
`File::writable_real?` (Class Method)

`write` `IO::write` (Class Method)

`yield` `Fiber::yield` (Class Method)

`zero?` `File::zero?` (Class Method)

`!` `BasicObject#!` (Instance Method)

`!=` `BasicObject#!=` (Instance Method)

`!~` `Object#!~` (Instance Method)

`%` `Float#%` (Instance Method)

`%` `Integer#%` (Instance Method)
`%` `Numeric#%` (Instance Method)
`%` `String#%` (Instance Method)
`&` `Array#&` (Instance Method)
`&` `FalseClass#&` (Instance Method)
`&` `Integer#&` (Instance Method)
`&` `NilClass#&` (Instance Method)
`&` `Process/Status#&` (Instance Method)
`&` `TrueClass#&` (Instance Method)
`*` `Array#*` (Instance Method)
`*` `Complex#*` (Instance Method)
`*` `Float#*` (Instance Method)
`*` `Integer#*` (Instance Method)
`*` `Rational#*` (Instance Method)
`*` `String#*` (Instance Method)
`**` `Complex#**` (Instance Method)
`**` `Float#**` (Instance Method)
`**` `Integer#**` (Instance Method)
`**` `Rational#**` (Instance Method)
`+` `Array#+` (Instance Method)
`+` `Complex#+` (Instance Method)
`+` `Float#+` (Instance Method)
`+` `Integer#+` (Instance Method)
`+` `Rational#+` (Instance Method)
`+` `String#+` (Instance Method)
`+` `Time#+` (Instance Method)
`+0` `Numeric#+0` (Instance Method)
`+0` `String#+0` (Instance Method)
`-` `Array#-` (Instance Method)
`-` `Complex#-` (Instance Method)
`-` `Float#-` (Instance Method)
`-` `Integer#-` (Instance Method)
`-` `Rational#-` (Instance Method)

`-` `Time#-` (Instance Method)
`-0` `Complex#-0` (Instance Method)
`-0` `Float#-0` (Instance Method)
`-0` `Integer#-0` (Instance Method)
`-0` `Numeric#-0` (Instance Method)
`-0` `Rational#-0` (Instance Method)
`-0` `String#-0` (Instance Method)
`/` `Complex#/` (Instance Method)
`/` `Float#/` (Instance Method)
`/` `Integer#/` (Instance Method)
`/` `Rational#/` (Instance Method)
`<` `Comparable#<` (Instance Method)
`<` `Float#<` (Instance Method)
`<` `Hash#<` (Instance Method)
`<` `Integer#<` (Instance Method)
`<` `Module#<` (Instance Method)
`<<` `Array#<<` (Instance Method)
`<<` `IO#<<` (Instance Method)
`<<` `Integer#<<` (Instance Method)
`<<` `Queue#<<` (Instance Method)
`<<` `SizedQueue#<<` (Instance Method)
`<<` `String#<<` (Instance Method)
`<=` `Comparable#<=` (Instance Method)
`<=` `Float#<=` (Instance Method)
`<=` `Hash#<=` (Instance Method)
`<=` `Integer#<=` (Instance Method)
`<=` `Module#<=` (Instance Method)
`<=>` `Array#<=>` (Instance Method)
`<=>` `File/Stat#<=>` (Instance Method)
`<=>` `Float#<=>` (Instance Method)
`<=>` `Integer#<=>` (Instance Method)
`<=>` `Module#<=>` (Instance Method)
`<=>` `Numeric#<=>` (Instance Method)

`===` `Proc#===` (Instance Method)
`===` `Range#===` (Instance Method)
`===` `Regexp#===` (Instance Method)
`===` `String#===` (Instance Method)
`===` `Symbol#===` (Instance Method)
`===` `TrueClass#===` (Instance Method)
`=~` `Object#=~` (Instance Method)
`=~` `Regexp#=~` (Instance Method)
`=~` `String#=~` (Instance Method)
`=~` `Symbol#=~` (Instance Method)
`>` `Comparable#>` (Instance Method)
`>` `Float#>` (Instance Method)
`>` `Hash#>` (Instance Method)
`>` `Integer#>` (Instance Method)
`>` `Module#>` (Instance Method)
`>=` `Comparable#>=` (Instance Method)
`>=` `Float#>=` (Instance Method)
`>=` `Hash#>=` (Instance Method)
`>=` `Integer#>=` (Instance Method)
`>=` `Module#>=` (Instance Method)
`>>` `Integer#>>` (Instance Method)
`>>` `Process/Status#>>` (Instance Method)
`Array` `Kernel#Array` (Instance Method)
`Complex` `Kernel#Complex` (Instance Method)
`Float` `Kernel#Float` (Instance Method)
`Hash` `Kernel#Hash` (Instance Method)
`Integer` `Kernel#Integer` (Instance Method)
`Rational` `Kernel#Rational` (Instance Method)
`String` `Kernel#String` (Instance Method)
`[]` `Array#[]` (Instance Method)
`[]` `Continuation#[]` (Instance Method)
`[]` `Hash#[]` (Instance Method)
`[]` `Integer#[]` (Instance Method)

abs `Numeric#abs` (Instance Method)

abs `Rational#abs` (Instance Method)

abs2 `Complex#abs2` (Instance Method)

abs2 `Numeric#abs2` (Instance Method)

absolute_path
`RubyVM/InstructionSequence#absolute_path` (Instance Method)

absolute_path
`Thread/Backtrace/Location#absolute_path` (Instance Method)

add `ThreadGroup#add` (Instance Method)

add_trace_func
`Thread#add_trace_func` (Instance Method)

advise `IO#advise` (Instance Method)

alias_method
`Module#alias_method` (Instance Method)

alive? `Fiber#alive?` (Instance Method)

alive? `Thread#alive?` (Instance Method)

all? `Enumerable#all?` (Instance Method)

allbits? `Integer#allbits?` (Instance Method)

allocate `Class#allocate` (Instance Method)

ancestors
`Module#ancestors` (Instance Method)

angle `Complex#angle` (Instance Method)

angle `Float#angle` (Instance Method)

angle `Numeric#angle` (Instance Method)

any? `Array#any?` (Instance Method)

any? `Enumerable#any?` (Instance Method)

any? `Hash#any?` (Instance Method)

anybits? `Integer#anybits?` (Instance Method)

append `Array#append` (Instance Method)

append_features
`Module#append_features` (Instance Method)

arg `Complex#arg` (Instance Method)

arg `Float#arg` (Instance Method)

arg `Numeric#arg` (Instance Method)

args `NoMethodError#args` (Instance Method)

backtrace
: `Thread#backtrace` (Instance Method)

backtrace_locations
: `Exception#backtrace_locations` (Instance Method)

backtrace_locations
: `Thread#backtrace_locations` (Instance Method)

base_label
: `RubyVM/InstructionSequence#base_label` (Instance Method)

base_label
: `Thread/Backtrace/Location#base_label` (Instance Method)

begin
: `MatchData#begin` (Instance Method)

begin
: `Range#begin` (Instance Method)

between?
: `Comparable#between?` (Instance Method)

bind
: `UnboundMethod#bind` (Instance Method)

binding
: `Kernel#binding` (Instance Method)

binding
: `Proc#binding` (Instance Method)

binding
: `TracePoint#binding` (Instance Method)

binmode
: `ARGF#binmode` (Instance Method)

binmode
: `IO#binmode` (Instance Method)

binmode?
: `ARGF#binmode?` (Instance Method)

binmode?
: `IO#binmode?` (Instance Method)

birthtime
: `File#birthtime` (Instance Method)

birthtime
: `File/Stat#birthtime` (Instance Method)

bit_length
: `Integer#bit_length` (Instance Method)

blksize
: `File/Stat#blksize` (Instance Method)

block_given?
: `Kernel#block_given?` (Instance Method)

blockdev?
: `File/Stat#blockdev?` (Instance Method)

blockdev?
: `FileTest#blockdev?` (Instance Method)

blocks
: `File/Stat#blocks` (Instance Method)

broadcast
: `ConditionVariable#broadcast` (Instance Method)

`ceil` `Float#ceil` (Instance Method)

`ceil` `Integer#ceil` (Instance Method)

`ceil` `Numeric#ceil` (Instance Method)

`ceil` `Rational#ceil` (Instance Method)

`center` `String#center` (Instance Method)

`chardev?` `File/Stat#chardev?` (Instance Method)

`chardev?` `FileTest#chardev?` (Instance Method)

`chars` `ARGF#chars` (Instance Method)

`chars` `IO#chars` (Instance Method)

`chars` `String#chars` (Instance Method)

`chmod` `File#chmod` (Instance Method)

`chomp` `Kernel#chomp` (Instance Method)

`chomp` `String#chomp` (Instance Method)

`chomp!` `String#chomp!` (Instance Method)

`chop` `Kernel#chop` (Instance Method)

`chop` `String#chop` (Instance Method)

`chop!` `String#chop!` (Instance Method)

`chown` `File#chown` (Instance Method)

`chr` `Integer#chr` (Instance Method)

`chr` `String#chr` (Instance Method)

`chunk` `Enumerable#chunk` (Instance Method)

`chunk` `Enumerator/Lazy#chunk` (Instance Method)

`chunk_while`
`Enumerable#chunk_while` (Instance Method)

`chunk_while`
`Enumerator/Lazy#chunk_while` (Instance Method)

`clamp` `Comparable#clamp` (Instance Method)

`class` `Object#class` (Instance Method)

`class_eval`
`Module#class_eval` (Instance Method)

`class_exec`
`Module#class_exec` (Instance Method)

`class_variable_defined?`
`Module#class_variable_defined?` (Instance Method)

class_variable_get
: Module#class_variable_get (Instance Method)

class_variable_set
: Module#class_variable_set (Instance Method)

class_variables
: Module#class_variables (Instance Method)

clear
: Array#clear (Instance Method)

clear
: Hash#clear (Instance Method)

clear
: Queue#clear (Instance Method)

clear
: SizedQueue#clear (Instance Method)

clear
: String#clear (Instance Method)

clone
: Method#clone (Instance Method)

clone
: Numeric#clone (Instance Method)

clone
: Object#clone (Instance Method)

clone
: UnboundMethod#clone (Instance Method)

close
: ARGF#close (Instance Method)

close
: Dir#close (Instance Method)

close
: IO#close (Instance Method)

close
: Queue#close (Instance Method)

close
: SizedQueue#close (Instance Method)

close_on_exec=
: IO#close_on_exec= (Instance Method)

close_on_exec?
: IO#close_on_exec? (Instance Method)

close_read
: IO#close_read (Instance Method)

close_write
: IO#close_write (Instance Method)

closed?
: ARGF#closed? (Instance Method)

closed?
: IO#closed? (Instance Method)

closed?
: Queue#closed? (Instance Method)

codepoints
: ARGF#codepoints (Instance Method)

codepoints
: IO#codepoints (Instance Method)

codepoints
: `String#codepoints` (Instance Method)

coerce
: `Float#coerce` (Instance Method)

coerce
: `Integer#coerce` (Instance Method)

coerce
: `Numeric#coerce` (Instance Method)

collect
: `Array#collect` (Instance Method)

collect
: `Enumerable#collect` (Instance Method)

collect
: `Enumerator/Lazy#collect` (Instance Method)

collect!
: `Array#collect!` (Instance Method)

collect_concat
: `Enumerable#collect_concat` (Instance Method)

collect_concat
: `Enumerator/Lazy#collect_concat` (Instance Method)

combination
: `Array#combination` (Instance Method)

compact
: `Array#compact` (Instance Method)

compact
: `Hash#compact` (Instance Method)

compact!
: `Array#compact!` (Instance Method)

compact!
: `Hash#compact!` (Instance Method)

compare_by_identity
: `Hash#compare_by_identity` (Instance Method)

compare_by_identity?
: `Hash#compare_by_identity?` (Instance Method)

concat
: `Array#concat` (Instance Method)

concat
: `String#concat` (Instance Method)

conj
: `Complex#conj` (Instance Method)

conj
: `Numeric#conj` (Instance Method)

conjugate
: `Complex#conjugate` (Instance Method)

conjugate
: `Numeric#conjugate` (Instance Method)

const_defined?
: `Module#const_defined?` (Instance Method)

const_get
: `Module#const_get` (Instance Method)

`delete` `Array#delete` (Instance Method)

`delete` `Hash#delete` (Instance Method)

`delete` `String#delete` (Instance Method)

`delete!` `String#delete!` (Instance Method)

`delete_at`
`Array#delete_at` (Instance Method)

`delete_if`
`Array#delete_if` (Instance Method)

`delete_if`
`Hash#delete_if` (Instance Method)

`delete_prefix`
`String#delete_prefix` (Instance Method)

`delete_prefix!`
`String#delete_prefix!` (Instance Method)

`delete_suffix`
`String#delete_suffix` (Instance Method)

`delete_suffix!`
`String#delete_suffix!` (Instance Method)

`denominator`
`Complex#denominator` (Instance Method)

`denominator`
`Float#denominator` (Instance Method)

`denominator`
`Integer#denominator` (Instance Method)

`denominator`
`Numeric#denominator` (Instance Method)

`denominator`
`Rational#denominator` (Instance Method)

`deprecate_constant`
`Module#deprecate_constant` (Instance Method)

`deq` `Queue#deq` (Instance Method)

`deq` `SizedQueue#deq` (Instance Method)

`destination_encoding`
`Encoding/Converter#destination_encoding` (Instance Method)

`destination_encoding`
`Encoding/InvalidByteSequenceError#destination_encoding` (Instance Method)

destination_encoding
: Encoding/UndefinedConversionError#destination_encoding (Instance Method)

destination_encoding_name
: Encoding/InvalidByteSequenceError#destination_encoding_name (Instance Method)

destination_encoding_name
: Encoding/UndefinedConversionError#destination_encoding_name (Instance Method)

detect
: Enumerable#detect (Instance Method)

dev
: File/Stat#dev (Instance Method)

dev_major
: File/Stat#dev_major (Instance Method)

dev_minor
: File/Stat#dev_minor (Instance Method)

dig
: Array#dig (Instance Method)

dig
: Hash#dig (Instance Method)

dig
: Struct#dig (Instance Method)

digits
: Integer#digits (Instance Method)

directory?
: File/Stat#directory? (Instance Method)

directory?
: FileTest#directory? (Instance Method)

disable
: TracePoint#disable (Instance Method)

disasm
: RubyVM/InstructionSequence#disasm (Instance Method)

disassemble
: RubyVM/InstructionSequence#disassemble (Instance Method)

display
: Object#display (Instance Method)

div
: Integer#div (Instance Method)

div
: Numeric#div (Instance Method)

divmod
: Float#divmod (Instance Method)

divmod
: Integer#divmod (Instance Method)

divmod
: Numeric#divmod (Instance Method)

downcase
: String#downcase (Instance Method)

downcase
: Symbol#downcase (Instance Method)

downcase!
: String#downcase! (Instance Method)

`downto` `Integer#downto` (Instance Method)

`drop` `Array#drop` (Instance Method)

`drop` `Enumerable#drop` (Instance Method)

`drop` `Enumerator/Lazy#drop` (Instance Method)

`drop_while` `Array#drop_while` (Instance Method)

`drop_while` `Enumerable#drop_while` (Instance Method)

`drop_while` `Enumerator/Lazy#drop_while` (Instance Method)

`dst?` `Time#dst?` (Instance Method)

`dummy?` `Encoding#dummy?` (Instance Method)

`dump` `String#dump` (Instance Method)

`dup` `Numeric#dup` (Instance Method)

`dup` `Object#dup` (Instance Method)

`each` `ARGF#each` (Instance Method)

`each` `Array#each` (Instance Method)

`each` `Dir#each` (Instance Method)

`each` `Enumerator#each` (Instance Method)

`each` `Hash#each` (Instance Method)

`each` `IO#each` (Instance Method)

`each` `ObjectSpace/WeakMap#each` (Instance Method)

`each` `Range#each` (Instance Method)

`each` `Struct#each` (Instance Method)

`each_byte` `ARGF#each_byte` (Instance Method)

`each_byte` `IO#each_byte` (Instance Method)

`each_byte` `String#each_byte` (Instance Method)

`each_char` `ARGF#each_char` (Instance Method)

`each_char` `IO#each_char` (Instance Method)

`each_char` `String#each_char` (Instance Method)

each_with_index
: `Enumerator#each_with_index` (Instance Method)

each_with_object
: `Enumerable#each_with_object` (Instance Method)

each_with_object
: `Enumerator#each_with_object` (Instance Method)

empty?
: `Array#empty?` (Instance Method)

empty?
: `FileTest#empty?` (Instance Method)

empty?
: `Hash#empty?` (Instance Method)

empty?
: `Queue#empty?` (Instance Method)

empty?
: `SizedQueue#empty?` (Instance Method)

empty?
: `String#empty?` (Instance Method)

empty?
: `Symbol#empty?` (Instance Method)

enable
: `TracePoint#enable` (Instance Method)

enabled?
: `TracePoint#enabled?` (Instance Method)

enclose
: `ThreadGroup#enclose` (Instance Method)

enclosed?
: `ThreadGroup#enclosed?` (Instance Method)

encode
: `String#encode` (Instance Method)

encode!
: `String#encode!` (Instance Method)

encoding
: `Regexp#encoding` (Instance Method)

encoding
: `String#encoding` (Instance Method)

encoding
: `Symbol#encoding` (Instance Method)

end
: `MatchData#end` (Instance Method)

end
: `Range#end` (Instance Method)

end_with?
: `String#end_with?` (Instance Method)

enq
: `Queue#enq` (Instance Method)

enq
: `SizedQueue#enq` (Instance Method)

entries
: `Enumerable#entries` (Instance Method)

enum_for
: `Enumerator/Lazy#enum_for` (Instance Method)

enum_for
: `Object#enum_for` (Instance Method)

eof
: `ARGF#eof` (Instance Method)

eof
: `IO#eof` (Instance Method)

eof? `ARGF#eof?` (Instance Method)

eof? `IO#eof?` (Instance Method)

eql? `Array#eql?` (Instance Method)

eql? `Float#eql?` (Instance Method)

eql? `Hash#eql?` (Instance Method)

eql? `MatchData#eql?` (Instance Method)

eql? `Method#eql?` (Instance Method)

eql? `Numeric#eql?` (Instance Method)

eql? `Object#eql?` (Instance Method)

eql? `Range#eql?` (Instance Method)

eql? `Regexp#eql?` (Instance Method)

eql? `String#eql?` (Instance Method)

eql? `Struct#eql?` (Instance Method)

eql? `Time#eql?` (Instance Method)

eql? `UnboundMethod#eql?` (Instance Method)

equal? `BasicObject#equal?` (Instance Method)

errno `SystemCallError#errno` (Instance Method)

error_bytes
`Encoding/InvalidByteSequenceError#error_bytes` (Instance Method)

error_char
`Encoding/UndefinedConversionError#error_char` (Instance Method)

eval `Binding#eval` (Instance Method)

eval `Kernel#eval` (Instance Method)

eval `RubyVM/InstructionSequence#eval` (Instance Method)

even? `Integer#even?` (Instance Method)

event `TracePoint#event` (Instance Method)

exception
`Exception#exception` (Instance Method)

exclude_end?
`Range#exclude_end?` (Instance Method)

exec `Kernel#exec` (Instance Method)

executable?
`File/Stat#executable?` (Instance Method)

executable?
`FileTest#executable?` (Instance Method)

executable_real?
: File/Stat#executable_real? (Instance Method)

executable_real?
: FileTest#executable_real? (Instance Method)

exist?
: FileTest#exist? (Instance Method)

exists?
: FileTest#exists? (Instance Method)

exit
: Kernel#exit (Instance Method)

exit
: Thread#exit (Instance Method)

exit!
: Kernel#exit! (Instance Method)

exit_value
: LocalJumpError#exit_value (Instance Method)

exited?
: Process/Status#exited? (Instance Method)

exitstatus
: Process/Status#exitstatus (Instance Method)

extend
: Object#extend (Instance Method)

extend_object
: Module#extend_object (Instance Method)

extended
: Module#extended (Instance Method)

external_encoding
: ARGF#external_encoding (Instance Method)

external_encoding
: IO#external_encoding (Instance Method)

fail
: Kernel#fail (Instance Method)

fcntl
: IO#fcntl (Instance Method)

fdatasync
: IO#fdatasync (Instance Method)

fdiv
: Complex#fdiv (Instance Method)

fdiv
: Float#fdiv (Instance Method)

fdiv
: Integer#fdiv (Instance Method)

fdiv
: Numeric#fdiv (Instance Method)

fdiv
: Rational#fdiv (Instance Method)

feed
: Enumerator#feed (Instance Method)

fetch
: Array#fetch (Instance Method)

fetch
: Hash#fetch (Instance Method)

fetch
: Thread#fetch (Instance Method)

fetch_values
: Hash#fetch_values (Instance Method)

file
: ARGF#file (Instance Method)

file?
: File/Stat#file? (Instance Method)

file?
: FileTest#file? (Instance Method)

filename
: ARGF#filename (Instance Method)

fileno
: ARGF#fileno (Instance Method)

fileno
: Dir#fileno (Instance Method)

fileno
: IO#fileno (Instance Method)

fill
: Array#fill (Instance Method)

finalize
: ObjectSpace/WeakMap#finalize (Instance Method)

find
: Enumerable#find (Instance Method)

find_all
: Enumerable#find_all (Instance Method)

find_all
: Enumerator/Lazy#find_all (Instance Method)

find_index
: Array#find_index (Instance Method)

find_index
: Enumerable#find_index (Instance Method)

finish
: Encoding/Converter#finish (Instance Method)

finite?
: Complex#finite? (Instance Method)

finite?
: Float#finite? (Instance Method)

finite?
: Numeric#finite? (Instance Method)

first
: Array#first (Instance Method)

first
: Enumerable#first (Instance Method)

first
: Range#first (Instance Method)

first_lineno
: RubyVM/InstructionSequence#first_lineno (Instance Method)

fixed_encoding?
: Regexp#fixed_encoding? (Instance Method)

flat_map
: Enumerable#flat_map (Instance Method)

flat_map
: Enumerator/Lazy#flat_map (Instance Method)

flatten
: Array#flatten (Instance Method)

flatten
: Hash#flatten (Instance Method)

flatten!
: Array#flatten! (Instance Method)

flock `File#flock` (Instance Method)

floor `Float#floor` (Instance Method)

floor `Integer#floor` (Instance Method)

floor `Numeric#floor` (Instance Method)

floor `Rational#floor` (Instance Method)

flush `IO#flush` (Instance Method)

force_encoding `String#force_encoding` (Instance Method)

fork `Kernel#fork` (Instance Method)

format `Kernel#format` (Instance Method)

freeze `Module#freeze` (Instance Method)

freeze `Object#freeze` (Instance Method)

freeze `String#freeze` (Instance Method)

friday? `Time#friday?` (Instance Method)

frozen? `Array#frozen?` (Instance Method)

frozen? `Object#frozen?` (Instance Method)

fsync `IO#fsync` (Instance Method)

ftype `File/Stat#ftype` (Instance Method)

full_message `Exception#full_message` (Instance Method)

garbage_collect `GC#garbage_collect` (Instance Method)

gcd `Integer#gcd` (Instance Method)

gcdlcm `Integer#gcdlcm` (Instance Method)

getbyte `ARGF#getbyte` (Instance Method)

getbyte `IO#getbyte` (Instance Method)

getbyte `String#getbyte` (Instance Method)

getc `ARGF#getc` (Instance Method)

getc `IO#getc` (Instance Method)

getgm `Time#getgm` (Instance Method)

getlocal `Time#getlocal` (Instance Method)

gets `ARGF#gets` (Instance Method)

gets `IO#gets` (Instance Method)

gets `Kernel#gets` (Instance Method)

getutc `Time#getutc` (Instance Method)

gid `File/Stat#gid` (Instance Method)

global_variables
`Kernel#global_variables` (Instance Method)

gmt? `Time#gmt?` (Instance Method)

gmt_offset
`Time#gmt_offset` (Instance Method)

gmtime `Time#gmtime` (Instance Method)

gmtoff `Time#gmtoff` (Instance Method)

grapheme_clusters
`String#grapheme_clusters` (Instance Method)

grep `Enumerable#grep` (Instance Method)

grep `Enumerator/Lazy#grep` (Instance Method)

grep_v `Enumerable#grep_v` (Instance Method)

grep_v `Enumerator/Lazy#grep_v` (Instance Method)

group `Thread#group` (Instance Method)

group_by `Enumerable#group_by` (Instance Method)

grpowned?
`File/Stat#grpowned?` (Instance Method)

grpowned?
`FileTest#grpowned?` (Instance Method)

gsub `Kernel#gsub` (Instance Method)

gsub `String#gsub` (Instance Method)

gsub! `String#gsub!` (Instance Method)

has_key? `Hash#has_key?` (Instance Method)

has_value?
`Hash#has_value?` (Instance Method)

hash `Array#hash` (Instance Method)

hash `Float#hash` (Instance Method)

hash `Hash#hash` (Instance Method)

hash `MatchData#hash` (Instance Method)

hash `Method#hash` (Instance Method)

hash `Proc#hash` (Instance Method)

hash `Range#hash` (Instance Method)

hash `Regexp#hash` (Instance Method)

hash `String#hash` (Instance Method)

hash `Struct#hash` (Instance Method)

hash `Time#hash` (Instance Method)

hash `UnboundMethod#hash` (Instance Method)

hex `String#hex` (Instance Method)

hour `Time#hour` (Instance Method)

i `Numeric#i` (Instance Method)

id2name `Symbol#id2name` (Instance Method)

identical? `FileTest#identical?` (Instance Method)

imag `Complex#imag` (Instance Method)

imag `Numeric#imag` (Instance Method)

imaginary `Complex#imaginary` (Instance Method)

imaginary `Numeric#imaginary` (Instance Method)

include `Module#include` (Instance Method)

include? `Array#include?` (Instance Method)

include? `Enumerable#include?` (Instance Method)

include? `Hash#include?` (Instance Method)

include? `Module#include?` (Instance Method)

include? `ObjectSpace/WeakMap#include?` (Instance Method)

include? `Range#include?` (Instance Method)

include? `String#include?` (Instance Method)

included `Module#included` (Instance Method)

included_modules `Module#included_modules` (Instance Method)

incomplete_input? `Encoding/InvalidByteSequenceError#incomplete_input?` (Instance Method)

index `Array#index` (Instance Method)

index `String#index` (Instance Method)

infinite? `Complex#infinite?` (Instance Method)

infinite? `Float#infinite?` (Instance Method)

infinite?
: `Numeric#infinite?` (Instance Method)

inherited
: `Class#inherited` (Instance Method)

initialize_copy
: `Array#initialize_copy` (Instance Method)

initialize_copy
: `String#initialize_copy` (Instance Method)

inject
: `Enumerable#inject` (Instance Method)

ino
: `File/Stat#ino` (Instance Method)

inplace_mode
: `ARGF#inplace_mode` (Instance Method)

inplace_mode=
: `ARGF#inplace_mode=` (Instance Method)

insert
: `Array#insert` (Instance Method)

insert
: `String#insert` (Instance Method)

insert_output
: `Encoding/Converter#insert_output` (Instance Method)

inspect
: `ARGF#inspect` (Instance Method)

inspect
: `Array#inspect` (Instance Method)

inspect
: `Complex#inspect` (Instance Method)

inspect
: `Dir#inspect` (Instance Method)

inspect
: `Encoding#inspect` (Instance Method)

inspect
: `Encoding/Converter#inspect` (Instance Method)

inspect
: `Enumerator#inspect` (Instance Method)

inspect
: `Exception#inspect` (Instance Method)

inspect
: `FalseClass#inspect` (Instance Method)

inspect
: `Fiber#inspect` (Instance Method)

inspect
: `File/Stat#inspect` (Instance Method)

inspect
: `Float#inspect` (Instance Method)

inspect
: `Hash#inspect` (Instance Method)

inspect
: `IO#inspect` (Instance Method)

inspect
: `Integer#inspect` (Instance Method)

inspect
: `MatchData#inspect` (Instance Method)

inspect
: `Method#inspect` (Instance Method)

inspect `Module#inspect` (Instance Method)

inspect `NilClass#inspect` (Instance Method)

inspect `Object#inspect` (Instance Method)

inspect `ObjectSpace/WeakMap#inspect` (Instance Method)

inspect `Proc#inspect` (Instance Method)

inspect `Process/Status#inspect` (Instance Method)

inspect `Range#inspect` (Instance Method)

inspect `Rational#inspect` (Instance Method)

inspect `Regexp#inspect` (Instance Method)

inspect `RubyVM/InstructionSequence#inspect` (Instance Method)

inspect `String#inspect` (Instance Method)

inspect `Struct#inspect` (Instance Method)

inspect `Symbol#inspect` (Instance Method)

inspect `Thread#inspect` (Instance Method)

inspect `Thread/Backtrace/Location#inspect` (Instance Method)

inspect `Time#inspect` (Instance Method)

inspect `TracePoint#inspect` (Instance Method)

inspect `TrueClass#inspect` (Instance Method)

inspect `UnboundMethod#inspect` (Instance Method)

instance_eval
`BasicObject#instance_eval` (Instance Method)

instance_exec
`BasicObject#instance_exec` (Instance Method)

instance_method
`Module#instance_method` (Instance Method)

instance_methods
`Module#instance_methods` (Instance Method)

instance_of?
`Object#instance_of?` (Instance Method)

instance_variable_defined?
`Object#instance_variable_defined?` (Instance Method)

instance_variable_get
`Object#instance_variable_get` (Instance Method)

instance_variable_set
`Object#instance_variable_set` (Instance Method)

`instance_variables`
: `Object#instance_variables` (Instance Method)

`integer?` `Integer#integer?` (Instance Method)

`integer?` `Numeric#integer?` (Instance Method)

`intern` `String#intern` (Instance Method)

`intern` `Symbol#intern` (Instance Method)

`internal_encoding`
: `ARGF#internal_encoding` (Instance Method)

`internal_encoding`
: `IO#internal_encoding` (Instance Method)

`invert` `Hash#invert` (Instance Method)

`ioctl` `IO#ioctl` (Instance Method)

`is_a?` `Object#is_a?` (Instance Method)

`isatty` `IO#isatty` (Instance Method)

`isdst` `Time#isdst` (Instance Method)

`iterator?`
: `Kernel#iterator?` (Instance Method)

`itself` `Object#itself` (Instance Method)

`join` `Array#join` (Instance Method)

`join` `Thread#join` (Instance Method)

`keep_if` `Array#keep_if` (Instance Method)

`keep_if` `Hash#keep_if` (Instance Method)

`key` `Hash#key` (Instance Method)

`key` `KeyError#key` (Instance Method)

`key?` `Hash#key?` (Instance Method)

`key?` `ObjectSpace/WeakMap#key?` (Instance Method)

`key?` `Thread#key?` (Instance Method)

`keys` `Hash#keys` (Instance Method)

`keys` `ObjectSpace/WeakMap#keys` (Instance Method)

`keys` `Thread#keys` (Instance Method)

`kill` `Thread#kill` (Instance Method)

`kind_of?` `Object#kind_of?` (Instance Method)

`label` `RubyVM/InstructionSequence#label` (Instance Method)

`label` `Thread/Backtrace/Location#label` (Instance Method)

`lambda` `Kernel#lambda` (Instance Method)

`lambda?` `Proc#lambda?` (Instance Method)

`last` `Array#last` (Instance Method)

`last` `Range#last` (Instance Method)

`last_error`
`Encoding/Converter#last_error` (Instance Method)

`lazy` `Enumerable#lazy` (Instance Method)

`lazy` `Enumerator/Lazy#lazy` (Instance Method)

`lcm` `Integer#lcm` (Instance Method)

`length` `Array#length` (Instance Method)

`length` `Hash#length` (Instance Method)

`length` `MatchData#length` (Instance Method)

`length` `ObjectSpace/WeakMap#length` (Instance Method)

`length` `Queue#length` (Instance Method)

`length` `SizedQueue#length` (Instance Method)

`length` `String#length` (Instance Method)

`length` `Struct#length` (Instance Method)

`length` `Symbol#length` (Instance Method)

`lineno` `ARGF#lineno` (Instance Method)

`lineno` `IO#lineno` (Instance Method)

`lineno` `Thread/Backtrace/Location#lineno` (Instance Method)

`lineno` `TracePoint#lineno` (Instance Method)

`lineno=` `ARGF#lineno=` (Instance Method)

`lineno=` `IO#lineno=` (Instance Method)

`lines` `ARGF#lines` (Instance Method)

`lines` `IO#lines` (Instance Method)

`lines` `String#lines` (Instance Method)

`list` `ThreadGroup#list` (Instance Method)

`ljust` `String#ljust` (Instance Method)

`load` `Kernel#load` (Instance Method)

`local_variable_defined?`
`Binding#local_variable_defined?` (Instance Method)

`local_variable_get`
`Binding#local_variable_get` (Instance Method)

`max` `Array#max` (Instance Method)

`max` `Enumerable#max` (Instance Method)

`max` `Range#max` (Instance Method)

`max` `SizedQueue#max` (Instance Method)

`max=` `SizedQueue#max=` (Instance Method)

`max_by` `Enumerable#max_by` (Instance Method)

`mday` `Time#mday` (Instance Method)

`member?` `Enumerable#member?` (Instance Method)

`member?` `Hash#member?` (Instance Method)

`member?` `ObjectSpace/WeakMap#member?` (Instance Method)

`member?` `Range#member?` (Instance Method)

`members` `Struct#members` (Instance Method)

`merge` `Hash#merge` (Instance Method)

`merge!` `Hash#merge!` (Instance Method)

`message` `Exception#message` (Instance Method)

`method` `Object#method` (Instance Method)

`method_added`
`Module#method_added` (Instance Method)

`method_defined?`
`Module#method_defined?` (Instance Method)

`method_id`
`TracePoint#method_id` (Instance Method)

`method_missing`
`BasicObject#method_missing` (Instance Method)

`method_removed`
`Module#method_removed` (Instance Method)

`method_undefined`
`Module#method_undefined` (Instance Method)

`methods` `Object#methods` (Instance Method)

`min` `Array#min` (Instance Method)

`min` `Enumerable#min` (Instance Method)

`min` `Range#min` (Instance Method)

`min` `Time#min` (Instance Method)

`min_by` `Enumerable#min_by` (Instance Method)

`minmax` `Enumerable#minmax` (Instance Method)

negative?
: `Rational#negative?` (Instance Method)

new
: `Class#new` (Instance Method)

next
: `Enumerator#next` (Instance Method)

next
: `Integer#next` (Instance Method)

next
: `String#next` (Instance Method)

next
: `Symbol#next` (Instance Method)

next!
: `String#next!` (Instance Method)

next_float
: `Float#next_float` (Instance Method)

next_values
: `Enumerator#next_values` (Instance Method)

nil?
: `NilClass#nil?` (Instance Method)

nil?
: `Object#nil?` (Instance Method)

nlink
: `File/Stat#nlink` (Instance Method)

nobits?
: `Integer#nobits?` (Instance Method)

none?
: `Enumerable#none?` (Instance Method)

nonzero?
: `Numeric#nonzero?` (Instance Method)

nsec
: `Time#nsec` (Instance Method)

num_waiting
: `Queue#num_waiting` (Instance Method)

num_waiting
: `SizedQueue#num_waiting` (Instance Method)

numerator
: `Complex#numerator` (Instance Method)

numerator
: `Float#numerator` (Instance Method)

numerator
: `Integer#numerator` (Instance Method)

numerator
: `Numeric#numerator` (Instance Method)

numerator
: `Rational#numerator` (Instance Method)

object_id
: `Object#object_id` (Instance Method)

oct
: `String#oct` (Instance Method)

odd? `Integer#odd?` (Instance Method)

offset `MatchData#offset` (Instance Method)

one? `Enumerable#one?` (Instance Method)

open `Kernel#open` (Instance Method)

options `Regexp#options` (Instance Method)

ord `Integer#ord` (Instance Method)

ord `String#ord` (Instance Method)

original_name `Method#original_name` (Instance Method)

original_name `UnboundMethod#original_name` (Instance Method)

owned? `File/Stat#owned?` (Instance Method)

owned? `FileTest#owned?` (Instance Method)

owned? `Mutex#owned?` (Instance Method)

owner `Method#owner` (Instance Method)

owner `UnboundMethod#owner` (Instance Method)

p `Kernel#p` (Instance Method)

pack `Array#pack` (Instance Method)

parameters `Method#parameters` (Instance Method)

parameters `Proc#parameters` (Instance Method)

parameters `UnboundMethod#parameters` (Instance Method)

partition `Enumerable#partition` (Instance Method)

partition `String#partition` (Instance Method)

path `ARGF#path` (Instance Method)

path `Dir#path` (Instance Method)

path `File#path` (Instance Method)

path `RubyVM/InstructionSequence#path` (Instance Method)

path `Thread/Backtrace/Location#path` (Instance Method)

path `TracePoint#path` (Instance Method)

peek `Enumerator#peek` (Instance Method)

peek_values
: `Enumerator#peek_values` (Instance Method)

pending_interrupt?
: `Thread#pending_interrupt?` (Instance Method)

permutation
: `Array#permutation` (Instance Method)

phase
: `Complex#phase` (Instance Method)

phase
: `Float#phase` (Instance Method)

phase
: `Numeric#phase` (Instance Method)

pid
: `IO#pid` (Instance Method)

pid
: `Process/Status#pid` (Instance Method)

pid
: `Process/Waiter#pid` (Instance Method)

pipe?
: `File/Stat#pipe?` (Instance Method)

pipe?
: `FileTest#pipe?` (Instance Method)

polar
: `Complex#polar` (Instance Method)

polar
: `Numeric#polar` (Instance Method)

pop
: `Array#pop` (Instance Method)

pop
: `Queue#pop` (Instance Method)

pop
: `SizedQueue#pop` (Instance Method)

pos
: `ARGF#pos` (Instance Method)

pos
: `Dir#pos` (Instance Method)

pos
: `IO#pos` (Instance Method)

pos=
: `ARGF#pos=` (Instance Method)

pos=
: `Dir#pos=` (Instance Method)

pos=
: `IO#pos=` (Instance Method)

positive?
: `Float#positive?` (Instance Method)

positive?
: `Numeric#positive?` (Instance Method)

positive?
: `Rational#positive?` (Instance Method)

post_match
: `MatchData#post_match` (Instance Method)

pow
: `Integer#pow` (Instance Method)

pre_match
: `MatchData#pre_match` (Instance Method)

pread `IO#pread` (Instance Method)

pred `Integer#pred` (Instance Method)

prepend `Array#prepend` (Instance Method)

prepend `Module#prepend` (Instance Method)

prepend `String#prepend` (Instance Method)

prepend_features
`Module#prepend_features` (Instance Method)

prepended
`Module#prepended` (Instance Method)

prev_float
`Float#prev_float` (Instance Method)

primitive_convert
`Encoding/Converter#primitive_convert` (Instance Method)

primitive_errinfo
`Encoding/Converter#primitive_errinfo` (Instance Method)

print `ARGF#print` (Instance Method)

print `IO#print` (Instance Method)

print `Kernel#print` (Instance Method)

printf `ARGF#printf` (Instance Method)

printf `IO#printf` (Instance Method)

printf `Kernel#printf` (Instance Method)

priority `Thread#priority` (Instance Method)

priority=
`Thread#priority=` (Instance Method)

private `Module#private` (Instance Method)

private_call?
`NoMethodError#private_call?` (Instance Method)

private_class_method
`Module#private_class_method` (Instance Method)

private_constant
`Module#private_constant` (Instance Method)

private_instance_methods
`Module#private_instance_methods` (Instance Method)

private_method_defined?
`Module#private_method_defined?` (Instance Method)

private_methods
`Object#private_methods` (Instance Method)

`proc` `Kernel#proc` (Instance Method)

`product` `Array#product` (Instance Method)

`protected`
`Module#protected` (Instance Method)

`protected_instance_methods`
`Module#protected_instance_methods` (Instance Method)

`protected_method_defined?`
`Module#protected_method_defined?` (Instance Method)

`protected_methods`
`Object#protected_methods` (Instance Method)

`public` `Module#public` (Instance Method)

`public_class_method`
`Module#public_class_method` (Instance Method)

`public_constant`
`Module#public_constant` (Instance Method)

`public_instance_method`
`Module#public_instance_method` (Instance Method)

`public_instance_methods`
`Module#public_instance_methods` (Instance Method)

`public_method`
`Object#public_method` (Instance Method)

`public_method_defined?`
`Module#public_method_defined?` (Instance Method)

`public_methods`
`Object#public_methods` (Instance Method)

`public_send`
`Object#public_send` (Instance Method)

`push` `Array#push` (Instance Method)

`push` `Queue#push` (Instance Method)

`push` `SizedQueue#push` (Instance Method)

`putback` `Encoding/Converter#putback` (Instance Method)

`putc` `ARGF#putc` (Instance Method)

`putc` `IO#putc` (Instance Method)

`putc` `Kernel#putc` (Instance Method)

`puts` `ARGF#puts` (Instance Method)

`puts` `IO#puts` (Instance Method)

puts `Kernel#puts` (Instance Method)

pwrite `IO#pwrite` (Instance Method)

quo `Complex#quo` (Instance Method)

quo `Float#quo` (Instance Method)

quo `Numeric#quo` (Instance Method)

quo `Rational#quo` (Instance Method)

raise `Kernel#raise` (Instance Method)

raise `Thread#raise` (Instance Method)

raised_exception
`TracePoint#raised_exception` (Instance Method)

rand `Kernel#rand` (Instance Method)

rand `Random#rand` (Instance Method)

rand `Random/Formatter#rand` (Instance Method)

random_number
`Random/Formatter#random_number` (Instance Method)

rassoc `Array#rassoc` (Instance Method)

rassoc `Hash#rassoc` (Instance Method)

rationalize
`Complex#rationalize` (Instance Method)

rationalize
`Float#rationalize` (Instance Method)

rationalize
`Integer#rationalize` (Instance Method)

rationalize
`NilClass#rationalize` (Instance Method)

rationalize
`Rational#rationalize` (Instance Method)

rdev `File/Stat#rdev` (Instance Method)

rdev_major
`File/Stat#rdev_major` (Instance Method)

rdev_minor
`File/Stat#rdev_minor` (Instance Method)

read `ARGF#read` (Instance Method)

read `Dir#read` (Instance Method)

read `IO#read` (Instance Method)

`read_nonblock`
: `ARGF#read_nonblock` (Instance Method)

`read_nonblock`
: `IO#read_nonblock` (Instance Method)

`readable?`
: `File/Stat#readable?` (Instance Method)

`readable?`
: `FileTest#readable?` (Instance Method)

`readable_real?`
: `File/Stat#readable_real?` (Instance Method)

`readable_real?`
: `FileTest#readable_real?` (Instance Method)

`readagain_bytes`
: `Encoding/InvalidByteSequenceError#readagain_bytes` (Instance Method)

`readbyte` `ARGF#readbyte` (Instance Method)

`readbyte` `IO#readbyte` (Instance Method)

`readchar` `ARGF#readchar` (Instance Method)

`readchar` `IO#readchar` (Instance Method)

`readline` `ARGF#readline` (Instance Method)

`readline` `IO#readline` (Instance Method)

`readline` `Kernel#readline` (Instance Method)

`readlines`
: `ARGF#readlines` (Instance Method)

`readlines`
: `IO#readlines` (Instance Method)

`readlines`
: `Kernel#readlines` (Instance Method)

`readpartial`
: `ARGF#readpartial` (Instance Method)

`readpartial`
: `IO#readpartial` (Instance Method)

`real` `Complex#real` (Instance Method)

`real` `Numeric#real` (Instance Method)

`real?` `Complex#real?` (Instance Method)

`real?` `Numeric#real?` (Instance Method)

`reason` `LocalJumpError#reason` (Instance Method)

`repeated_permutation`
: `Array#repeated_permutation` (Instance Method)

`replace`
: `Array#replace` (Instance Method)

`replace`
: `Hash#replace` (Instance Method)

`replace`
: `String#replace` (Instance Method)

`replacement`
: `Encoding/Converter#replacement` (Instance Method)

`replacement=`
: `Encoding/Converter#replacement=` (Instance Method)

`replicate`
: `Encoding#replicate` (Instance Method)

`report_on_exception`
: `Thread#report_on_exception` (Instance Method)

`report_on_exception=`
: `Thread#report_on_exception=` (Instance Method)

`require`
: `Kernel#require` (Instance Method)

`require_relative`
: `Kernel#require_relative` (Instance Method)

`respond_to?`
: `Object#respond_to?` (Instance Method)

`respond_to_missing?`
: `Object#respond_to_missing?` (Instance Method)

`result`
: `StopIteration#result` (Instance Method)

`resume`
: `Fiber#resume` (Instance Method)

`return_value`
: `TracePoint#return_value` (Instance Method)

`reverse`
: `Array#reverse` (Instance Method)

`reverse`
: `String#reverse` (Instance Method)

`reverse!`
: `Array#reverse!` (Instance Method)

`reverse!`
: `String#reverse!` (Instance Method)

`reverse_each`
: `Array#reverse_each` (Instance Method)

`reverse_each`
: `Enumerable#reverse_each` (Instance Method)

`rewind`
: `ARGF#rewind` (Instance Method)

`rewind`
: `Dir#rewind` (Instance Method)

rewind `Enumerator#rewind` (Instance Method)

rewind `IO#rewind` (Instance Method)

rindex `Array#rindex` (Instance Method)

rindex `String#rindex` (Instance Method)

rjust `String#rjust` (Instance Method)

rotate `Array#rotate` (Instance Method)

rotate! `Array#rotate!` (Instance Method)

round `Float#round` (Instance Method)

round `Integer#round` (Instance Method)

round `Numeric#round` (Instance Method)

round `Rational#round` (Instance Method)

round `Time#round` (Instance Method)

rpartition `String#rpartition` (Instance Method)

rstrip `String#rstrip` (Instance Method)

rstrip! `String#rstrip!` (Instance Method)

run `Thread#run` (Instance Method)

safe_level `Thread#safe_level` (Instance Method)

sample `Array#sample` (Instance Method)

saturday? `Time#saturday?` (Instance Method)

scan `String#scan` (Instance Method)

scrub `String#scrub` (Instance Method)

scrub! `String#scrub!` (Instance Method)

sec `Time#sec` (Instance Method)

seed `Random#seed` (Instance Method)

seek `ARGF#seek` (Instance Method)

seek `Dir#seek` (Instance Method)

seek `IO#seek` (Instance Method)

select `Array#select` (Instance Method)

select `Enumerable#select` (Instance Method)

select `Enumerator/Lazy#select` (Instance Method)

select `Hash#select` (Instance Method)

`select` `Kernel#select` (Instance Method)

`select` `Struct#select` (Instance Method)

`select!` `Array#select!` (Instance Method)

`select!` `Hash#select!` (Instance Method)

`self` `TracePoint#self` (Instance Method)

`send` `Object#send` (Instance Method)

`set_backtrace`
`Exception#set_backtrace` (Instance Method)

`set_encoding`
`ARGF#set_encoding` (Instance Method)

`set_encoding`
`IO#set_encoding` (Instance Method)

`set_trace_func`
`Kernel#set_trace_func` (Instance Method)

`set_trace_func`
`Thread#set_trace_func` (Instance Method)

`setbyte` `String#setbyte` (Instance Method)

`setgid?` `File/Stat#setgid?` (Instance Method)

`setgid?` `FileTest#setgid?` (Instance Method)

`setuid?` `File/Stat#setuid?` (Instance Method)

`setuid?` `FileTest#setuid?` (Instance Method)

`shift` `Array#shift` (Instance Method)

`shift` `Hash#shift` (Instance Method)

`shift` `Queue#shift` (Instance Method)

`shift` `SizedQueue#shift` (Instance Method)

`shuffle` `Array#shuffle` (Instance Method)

`shuffle!` `Array#shuffle!` (Instance Method)

`signal` `ConditionVariable#signal` (Instance Method)

`signaled?`
`Process/Status#signaled?` (Instance Method)

`signo` `SignalException#signo` (Instance Method)

`singleton_class`
`Object#singleton_class` (Instance Method)

`singleton_class?`
`Module#singleton_class?` (Instance Method)

`singleton_method`
: `Object#singleton_method` (Instance Method)

`singleton_method_added`
: `BasicObject#singleton_method_added` (Instance Method)

`singleton_method_removed`
: `BasicObject#singleton_method_removed` (Instance Method)

`singleton_method_undefined`
: `BasicObject#singleton_method_undefined` (Instance Method)

`singleton_methods`
: `Object#singleton_methods` (Instance Method)

`size` `Array#size` (Instance Method)

`size` `Enumerator#size` (Instance Method)

`size` `File#size` (Instance Method)

`size` `File/Stat#size` (Instance Method)

`size` `FileTest#size` (Instance Method)

`size` `Hash#size` (Instance Method)

`size` `Integer#size` (Instance Method)

`size` `MatchData#size` (Instance Method)

`size` `ObjectSpace/WeakMap#size` (Instance Method)

`size` `Queue#size` (Instance Method)

`size` `Range#size` (Instance Method)

`size` `SizedQueue#size` (Instance Method)

`size` `String#size` (Instance Method)

`size` `Struct#size` (Instance Method)

`size` `Symbol#size` (Instance Method)

`size?` `File/Stat#size?` (Instance Method)

`size?` `FileTest#size?` (Instance Method)

`skip` `ARGF#skip` (Instance Method)

`sleep` `Kernel#sleep` (Instance Method)

`sleep` `Mutex#sleep` (Instance Method)

`slice` `Array#slice` (Instance Method)

`slice` `Hash#slice` (Instance Method)

`slice` `String#slice` (Instance Method)

`slice` `Symbol#slice` (Instance Method)

slice! `Array#slice!` (Instance Method)

slice! `String#slice!` (Instance Method)

slice_after
`Enumerable#slice_after` (Instance Method)

slice_after
`Enumerator/Lazy#slice_after` (Instance Method)

slice_before
`Enumerable#slice_before` (Instance Method)

slice_before
`Enumerator/Lazy#slice_before` (Instance Method)

slice_when
`Enumerable#slice_when` (Instance Method)

slice_when
`Enumerator/Lazy#slice_when` (Instance Method)

socket? `File/Stat#socket?` (Instance Method)

socket? `FileTest#socket?` (Instance Method)

sort `Array#sort` (Instance Method)

sort `Enumerable#sort` (Instance Method)

sort! `Array#sort!` (Instance Method)

sort_by `Enumerable#sort_by` (Instance Method)

sort_by! `Array#sort_by!` (Instance Method)

source `Regexp#source` (Instance Method)

source_encoding
`Encoding/Converter#source_encoding` (Instance Method)

source_encoding
`Encoding/InvalidByteSequenceError#source_encoding` (Instance Method)

source_encoding
`Encoding/UndefinedConversionError#source_encoding` (Instance Method)

source_encoding_name
`Encoding/InvalidByteSequenceError#source_encoding_name` (Instance Method)

source_encoding_name
`Encoding/UndefinedConversionError#source_encoding_name` (Instance Method)

source_location
`Method#source_location` (Instance Method)

source_location
`Proc#source_location` (Instance Method)

source_location
: `UnboundMethod#source_location` (Instance Method)

spawn
: `Kernel#spawn` (Instance Method)

split
: `String#split` (Instance Method)

sprintf
: `Kernel#sprintf` (Instance Method)

squeeze
: `String#squeeze` (Instance Method)

squeeze!
: `String#squeeze!` (Instance Method)

srand
: `Kernel#srand` (Instance Method)

start_with?
: `String#start_with?` (Instance Method)

stat
: `IO#stat` (Instance Method)

status
: `SystemExit#status` (Instance Method)

status
: `Thread#status` (Instance Method)

step
: `Numeric#step` (Instance Method)

step
: `Range#step` (Instance Method)

sticky?
: `File/Stat#sticky?` (Instance Method)

sticky?
: `FileTest#sticky?` (Instance Method)

stop?
: `Thread#stop?` (Instance Method)

stopped?
: `Process/Status#stopped?` (Instance Method)

stopsig
: `Process/Status#stopsig` (Instance Method)

store
: `Hash#store` (Instance Method)

strftime
: `Time#strftime` (Instance Method)

string
: `MatchData#string` (Instance Method)

strip
: `String#strip` (Instance Method)

strip!
: `String#strip!` (Instance Method)

sub
: `Kernel#sub` (Instance Method)

sub
: `String#sub` (Instance Method)

sub!
: `String#sub!` (Instance Method)

subsec
: `Time#subsec` (Instance Method)

succ
: `Integer#succ` (Instance Method)

succ
: `String#succ` (Instance Method)

succ
: `Symbol#succ` (Instance Method)

succ
: `Time#succ` (Instance Method)

succ! `String#succ!` (Instance Method)

success? `Process/Status#success?` (Instance Method)

success? `SystemExit#success?` (Instance Method)

sum `Array#sum` (Instance Method)

sum `Enumerable#sum` (Instance Method)

sum `String#sum` (Instance Method)

sunday? `Time#sunday?` (Instance Method)

super_method
`Method#super_method` (Instance Method)

super_method
`UnboundMethod#super_method` (Instance Method)

superclass
`Class#superclass` (Instance Method)

swapcase `String#swapcase` (Instance Method)

swapcase `Symbol#swapcase` (Instance Method)

swapcase!
`String#swapcase!` (Instance Method)

symlink? `File/Stat#symlink?` (Instance Method)

symlink? `FileTest#symlink?` (Instance Method)

sync `IO#sync` (Instance Method)

sync= `IO#sync=` (Instance Method)

synchronize
`Mutex#synchronize` (Instance Method)

syscall `Kernel#syscall` (Instance Method)

sysread `IO#sysread` (Instance Method)

sysseek `IO#sysseek` (Instance Method)

system `Kernel#system` (Instance Method)

syswrite `IO#syswrite` (Instance Method)

tag `UncaughtThrowError#tag` (Instance Method)

taint `Object#taint` (Instance Method)

tainted? `Object#tainted?` (Instance Method)

take `Array#take` (Instance Method)

take `Enumerable#take` (Instance Method)

take `Enumerator/Lazy#take` (Instance Method)

take_while
: `Array#take_while` (Instance Method)

take_while
: `Enumerable#take_while` (Instance Method)

take_while
: `Enumerator/Lazy#take_while` (Instance Method)

tap
: `Object#tap` (Instance Method)

tell
: `ARGF#tell` (Instance Method)

tell
: `Dir#tell` (Instance Method)

tell
: `IO#tell` (Instance Method)

terminate
: `Thread#terminate` (Instance Method)

termsig
: `Process/Status#termsig` (Instance Method)

test
: `Kernel#test` (Instance Method)

thread_variable?
: `Thread#thread_variable?` (Instance Method)

thread_variable_get
: `Thread#thread_variable_get` (Instance Method)

thread_variable_set
: `Thread#thread_variable_set` (Instance Method)

thread_variables
: `Thread#thread_variables` (Instance Method)

throw
: `Kernel#throw` (Instance Method)

thursday?
: `Time#thursday?` (Instance Method)

times
: `Integer#times` (Instance Method)

to_a
: `ARGF#to_a` (Instance Method)

to_a
: `Array#to_a` (Instance Method)

to_a
: `Enumerable#to_a` (Instance Method)

to_a
: `Hash#to_a` (Instance Method)

to_a
: `MatchData#to_a` (Instance Method)

to_a
: `NilClass#to_a` (Instance Method)

to_a
: `RubyVM/InstructionSequence#to_a` (Instance Method)

to_a
: `Struct#to_a` (Instance Method)

to_a
: `Time#to_a` (Instance Method)

to_ary `Array#to_ary` (Instance Method)

to_binary `RubyVM/InstructionSequence#to_binary` (Instance Method)

to_c `Complex#to_c` (Instance Method)

to_c `NilClass#to_c` (Instance Method)

to_c `Numeric#to_c` (Instance Method)

to_c `String#to_c` (Instance Method)

to_enum `Enumerator/Lazy#to_enum` (Instance Method)

to_enum `Object#to_enum` (Instance Method)

to_f `Complex#to_f` (Instance Method)

to_f `Float#to_f` (Instance Method)

to_f `Integer#to_f` (Instance Method)

to_f `NilClass#to_f` (Instance Method)

to_f `Rational#to_f` (Instance Method)

to_f `String#to_f` (Instance Method)

to_f `Time#to_f` (Instance Method)

to_h `Array#to_h` (Instance Method)

to_h `Enumerable#to_h` (Instance Method)

to_h `Hash#to_h` (Instance Method)

to_h `NilClass#to_h` (Instance Method)

to_h `Struct#to_h` (Instance Method)

to_hash `Hash#to_hash` (Instance Method)

to_i `ARGF#to_i` (Instance Method)

to_i `Complex#to_i` (Instance Method)

to_i `Float#to_i` (Instance Method)

to_i `IO#to_i` (Instance Method)

to_i `Integer#to_i` (Instance Method)

to_i `NilClass#to_i` (Instance Method)

to_i `Process/Status#to_i` (Instance Method)

to_i `Rational#to_i` (Instance Method)

to_i `String#to_i` (Instance Method)

to_i `Time#to_i` (Instance Method)

to_int `Float#to_int` (Instance Method)

to_int Integer#to_int (Instance Method)
to_int Numeric#to_int (Instance Method)
to_io ARGF#to_io (Instance Method)
to_io IO#to_io (Instance Method)
to_path Dir#to_path (Instance Method)
to_path File#to_path (Instance Method)
to_proc Hash#to_proc (Instance Method)
to_proc Method#to_proc (Instance Method)
to_proc Proc#to_proc (Instance Method)
to_proc Symbol#to_proc (Instance Method)
to_r Complex#to_r (Instance Method)
to_r Float#to_r (Instance Method)
to_r Integer#to_r (Instance Method)
to_r NilClass#to_r (Instance Method)
to_r Rational#to_r (Instance Method)
to_r String#to_r (Instance Method)
to_r Time#to_r (Instance Method)
to_s ARGF#to_s (Instance Method)
to_s Array#to_s (Instance Method)
to_s Complex#to_s (Instance Method)
to_s Encoding#to_s (Instance Method)
to_s Exception#to_s (Instance Method)
to_s FalseClass#to_s (Instance Method)
to_s Fiber#to_s (Instance Method)
to_s Float#to_s (Instance Method)
to_s Hash#to_s (Instance Method)
to_s Integer#to_s (Instance Method)
to_s MatchData#to_s (Instance Method)
to_s Method#to_s (Instance Method)
to_s Module#to_s (Instance Method)
to_s NilClass#to_s (Instance Method)
to_s Object#to_s (Instance Method)
to_s Proc#to_s (Instance Method)

to_s Process/Status#to_s (Instance Method)

to_s Range#to_s (Instance Method)

to_s Rational#to_s (Instance Method)

to_s Regexp#to_s (Instance Method)

to_s String#to_s (Instance Method)

to_s Struct#to_s (Instance Method)

to_s Symbol#to_s (Instance Method)

to_s Thread#to_s (Instance Method)

to_s Thread/Backtrace/Location#to_s (Instance Method)

to_s Time#to_s (Instance Method)

to_s TrueClass#to_s (Instance Method)

to_s UnboundMethod#to_s (Instance Method)

to_s UncaughtThrowError#to_s (Instance Method)

to_str String#to_str (Instance Method)

to_sym String#to_sym (Instance Method)

to_sym Symbol#to_sym (Instance Method)

to_write_io
ARGF#to_write_io (Instance Method)

tr String#tr (Instance Method)

tr! String#tr! (Instance Method)

tr_s String#tr_s (Instance Method)

tr_s! String#tr_s! (Instance Method)

trace_points
RubyVM/InstructionSequence#trace_points (Instance Method)

trace_var
Kernel#trace_var (Instance Method)

transfer Fiber#transfer (Instance Method)

transform_keys
Hash#transform_keys (Instance Method)

transform_keys!
Hash#transform_keys! (Instance Method)

transform_values
Hash#transform_values (Instance Method)

transform_values!
Hash#transform_values! (Instance Method)

transpose
: `Array#transpose` (Instance Method)

trap
: `Kernel#trap` (Instance Method)

truncate
: `File#truncate` (Instance Method)

truncate
: `Float#truncate` (Instance Method)

truncate
: `Integer#truncate` (Instance Method)

truncate
: `Numeric#truncate` (Instance Method)

truncate
: `Rational#truncate` (Instance Method)

trust
: `Object#trust` (Instance Method)

try_lock
: `Mutex#try_lock` (Instance Method)

tty?
: `IO#tty?` (Instance Method)

tuesday?
: `Time#tuesday?` (Instance Method)

tv_nsec
: `Time#tv_nsec` (Instance Method)

tv_sec
: `Time#tv_sec` (Instance Method)

tv_usec
: `Time#tv_usec` (Instance Method)

uid
: `File/Stat#uid` (Instance Method)

unbind
: `Method#unbind` (Instance Method)

undef_method
: `Module#undef_method` (Instance Method)

undump
: `String#undump` (Instance Method)

ungetbyte
: `IO#ungetbyte` (Instance Method)

ungetc
: `IO#ungetc` (Instance Method)

unicode_normalize
: `String#unicode_normalize` (Instance Method)

unicode_normalize!
: `String#unicode_normalize!` (Instance Method)

unicode_normalized?
: `String#unicode_normalized?` (Instance Method)

uniq
: `Array#uniq` (Instance Method)

uniq
: `Enumerable#uniq` (Instance Method)

uniq
: `Enumerator/Lazy#uniq` (Instance Method)

uniq!
: `Array#uniq!` (Instance Method)

unlock
: `Mutex#unlock` (Instance Method)

unpack
: `String#unpack` (Instance Method)

`unpack1` `String#unpack1` (Instance Method)

`unshift` `Array#unshift` (Instance Method)

`untaint` `Object#untaint` (Instance Method)

`untrace_var`
`Kernel#untrace_var` (Instance Method)

`untrust` `Object#untrust` (Instance Method)

`untrusted?`
`Object#untrusted?` (Instance Method)

`upcase` `String#upcase` (Instance Method)

`upcase` `Symbol#upcase` (Instance Method)

`upcase!` `String#upcase!` (Instance Method)

`update` `Hash#update` (Instance Method)

`upto` `Integer#upto` (Instance Method)

`upto` `String#upto` (Instance Method)

`usec` `Time#usec` (Instance Method)

`using` `Module#using` (Instance Method)

`utc` `Time#utc` (Instance Method)

`utc?` `Time#utc?` (Instance Method)

`utc_offset`
`Time#utc_offset` (Instance Method)

`valid_encoding?`
`String#valid_encoding?` (Instance Method)

`value` `Thread#value` (Instance Method)

`value` `UncaughtThrowError#value` (Instance Method)

`value?` `Hash#value?` (Instance Method)

`values` `Hash#values` (Instance Method)

`values` `ObjectSpace/WeakMap#values` (Instance Method)

`values` `Struct#values` (Instance Method)

`values_at`
`Array#values_at` (Instance Method)

`values_at`
`Hash#values_at` (Instance Method)

`values_at`
`MatchData#values_at` (Instance Method)

`values_at`
`Struct#values_at` (Instance Method)

wait ConditionVariable#wait (Instance Method)

wakeup Thread#wakeup (Instance Method)

warn Kernel#warn (Instance Method)

warn Warning#warn (Instance Method)

wday Time#wday (Instance Method)

wednesday?
Time#wednesday? (Instance Method)

with_index
Enumerator#with_index (Instance Method)

with_object
Enumerator#with_object (Instance Method)

world_readable?
File/Stat#world_readable? (Instance Method)

world_readable?
FileTest#world_readable? (Instance Method)

world_writable?
File/Stat#world_writable? (Instance Method)

world_writable?
FileTest#world_writable? (Instance Method)

writable?
File/Stat#writable? (Instance Method)

writable?
FileTest#writable? (Instance Method)

writable_real?
File/Stat#writable_real? (Instance Method)

writable_real?
FileTest#writable_real? (Instance Method)

write ARGF#write (Instance Method)

write IO#write (Instance Method)

write Warning/buffer#write (Instance Method)

write_nonblock
IO#write_nonblock (Instance Method)

yday Time#yday (Instance Method)

year Time#year (Instance Method)

yield Proc#yield (Instance Method)

yield_self
Object#yield_self (Instance Method)

zero?	`File/Stat#zero?` (Instance Method)
zero?	`FileTest#zero?` (Instance Method)
zero?	`Float#zero?` (Instance Method)
zero?	`Numeric#zero?` (Instance Method)
zip	`Array#zip` (Instance Method)
zip	`Enumerable#zip` (Instance Method)
zip	`Enumerator/Lazy#zip` (Instance Method)
zone	`Time#zone` (Instance Method)
|	`Array#|` (Instance Method)
|	`FalseClass#|` (Instance Method)
|	`Integer#|` (Instance Method)
|	`NilClass#|` (Instance Method)
|	`TrueClass#|` (Instance Method)
~	`Complex#~` (Instance Method)
~	`Integer#~` (Instance Method)
~	`Regexp#~` (Instance Method)

A.1.4 Beginner Core Topics

Syntax http://ruby-doc.org/core-2.5.1/doc/syntax_rdoc.html

Control Expressions
http://ruby-doc.org/core-2.5.1/doc/syntax/control_expressions_rdoc.html

Assignment
http://ruby-doc.org/core-2.5.1/doc/syntax/assignment_rdoc.html

Methods http://ruby-doc.org/core-2.5.1/doc/syntax/methods_rdoc.html

Modules and Classes
http://ruby-doc.org/core-2.5.1/doc/syntax/modules_and_classes_rdoc.html

Operator Precedence
http://ruby-doc.org/core-2.5.1/doc/syntax/precedence_rdoc.html

Appendix B Ruby Gems

Ruby Gems
Ruby Gems Github Home

About

'`RubyGems.org`' is the Ruby communitys gem hosting service. Instantly publish your gems and then install them. Use the API to find out more about available gems. Become a contributor and improve the site yourself.

Guides

Learn how RubyGems works, and how to make your own.

The `RubyGems` software allows you to easily download, install, and use ruby software packages on your system. The software package is called a *gem* which contains a packaged Ruby application or library.

Gems can be used to extend or modify functionality in Ruby applications. Commonly theyre used to distribute reusable functionality that is shared with other Rubyists for use in their applications and libraries. Some gems provide command line utilities to help automate tasks and speed up your work.

B.1 Ruby Gem Guides

Guides

- RUBYGEMS BASICS
- WHAT IS A GEM?
- MAKE YOUR OWN GEM
- GEMS WITH EXTENSIONS
- NAME YOUR GEM
- PUBLISHING YOUR GEM
- SECURITY PRACTICES
- SSL CERTIFICATE UPDATE
- PATTERNS
- SPECIFICATION REFERENCE
- COMMAND REFERENCE
- RUBYGEMS API
- RUBYGEMS.ORG API
- RUBYGEMS.ORG API V2.0
- RUN YOUR OWN GEM SERVER
- RESOURCES
- CONTRIBUTING TO RUBYGEMS
- FREQUENTLY ASKED QUESTIONS

- PLUGINS
- CREDITS
- COMMON VULNERABILITIES AND EXPOSURES
- RELEASING RUBYGEMS

B.1.1 Ruby Gems Basics

The `gem` command allows you to interact with RubyGems.

Ruby 1.9 and newer ships with RubyGems built-in. Download RubyGems. To upgrade to the latest RubyGems: '`$ gem update --system`'.

B.1.2 What Is A Gem?

STRUCTURE OF A GEM

Each gem has a

- name,
- version, and
- platform.

For example, the `rake` gem has a 0.8.7 version (from May, 2009). `Rake`s platform is ruby, which means it works on any platform Ruby runs on.

Platforms

Platforms are based on the CPU architecture, operating system type and sometimes the operating system version. Examples include '`x86-mingw32`' or '`java`'. The platform indicates the gem only works with a ruby built for the same platform. RubyGems will automatically download the correct version for your platform. See `gem help platform` for full detail.

Components

Inside gems are the following components:

- Code (including tests and supporting utilities)
- Documentation
- gemspec

Code Organization

Each gem follows the same standard structure of code organization:

```
% tree freewill
freewill/
 bin/
    freewill
 lib/
    freewill.rb
 test/
    test_freewill.rb
 README
```

```
Rakefile
freewill.gemspec
```

Here, you can see the major components of a gem:

`lib`	The lib directory contains the code for the gem
`test`	The test or spec directory contains tests, depending on which test framework the developer uses
`Rakefile`	A gem usually has a `Rakefile`, which the `rake` program uses to automate tests, generate code, and perform other tasks.
`bin`	This gem also includes an executable file in the `bin` directory, which will be loaded into the users `PATH` when the gem is installed.
`README`	Documentation is usually included in the `README` and inline with the code. When you install a gem, documentation is generated automatically for you. Most gems include RDoc documentation, but some use YARD docs instead.
`gemspec`	The final piece is the gemspec, which contains information about the gem. The gems files, test information, platform, version number and more are all laid out here along with the authors email and name.

Table B.1: Table of RubyGem Components

The Gemspec

The gemspec specifies the information about a gem such as its name, version, description, authors and homepage.

Heres an example of a gemspec file.

```
% cat freewill.gemspec
Gem::Specification.new do |s|
  s.name        = 'freewill'
  s.version     = '1.0.0'
  s.summary     = "Freewill!"
  s.description = "I will choose Freewill!"
  s.authors     = ["Nick Quaranto"]
  s.email       = 'nick@quaran.to'
  s.homepage    = 'http://example.com/freewill'
  s.files       = ["lib/freewill.rb", ...]
end
```

B.1.3 Make a Gem

NOTE: Many people use Bundler to create Gems. You can learn how to do that by reading the *Developing a RubyGem using Bundler* guide on the Bundler website.

Introduction to Making a Gem

Creating and publishing your own gem is simple thanks to the tools baked right into RubyGems. Lets make a simple "hello world" gem, and feel free to play along at home! The code for the gem were going to make here is up on GitHub Hello World.

B.1.3.1 Your First Gem

I started with just one Ruby file for my `hola` gem, and the gemspec. Youll need a new name for yours (maybe `hola_yourusername`) to publish it. Check the Patterns guide for basic recommendations to follow when naming a gem.

```
% tree
.
 hola.gemspec
 lib
     hola.rb
```

`lib` Directory

Code for your package is placed within the `lib` directory. The convention is to have one Ruby file with the same name as your gem, since that gets loaded when `require 'hola'` is run. That one file is in charge of setting up your gems code and API.

`hola.rb`

The code inside of `lib/hola.rb` is pretty bare bones. It just makes sure that you can see some output from the gem:

{`hola.rb`} ≡

```
class Hola
  def self.hi
    puts "Hello world!"
  end
end
```

```
@post_create hola.rb mkdir -p src/gems/hola/lib/ && mv hola.rb src/gems/hola/lib
```

`hola.gemspec`

The gemspec defines whats in the gem, who made it, and the version of the gem. Its also your interface to '`RubyGems.org`'. All of the information you see on a gem page (like jekylls) comes from the gemspec.

{`hola.gemspec`} ≡

```
Gem::Specification.new do |s|
  s.name        = 'hola'
  s.version     = '0.0.0'
  s.date        = '2010-04-28'
  s.summary     = "Hola!"
  s.description = "A simple hello world gem"
  s.authors     = ["Nick Quaranto"]
  s.email       = 'nick@quaran.to'
```

```
  s.files       = ["lib/hola.rb"]
  s.homepage    =
    'http://rubygems.org/gems/hola'
  s.license       = 'MIT'
end
```

```
@post_create hola.gemspec mv hola.gemspec src/gems/hola/ && cd src/gems/hola
```

NOTE: The `description` member can be much longer than you see in this example. If it matches '`/^== [A-Z]/`' then the description will be run through RDocs markup formatter for display on the RubyGems web site. Be aware though that other consumers of the data might not understand this markup.

Look familiar? The gemspec is also Ruby, so you can wrap scripts to generate the file names and bump the version number. There are lots of fields the gemspec can contain. To see them all check out the full reference.

`build` and `install` A Gem From The Gemspec

After you have created a gemspec, you can `build` a gem from it. Then you can `install` the generated gem locally to test it out.

```
% gem build hola.gemspec
Successfully built RubyGem
Name: hola
Version: 0.0.0
File: hola-0.0.0.gem

% gem install ./hola-0.0.0.gem
Successfully installed hola-0.0.0
1 gem installed
```

The final step is to `require` the gem and use it:

```
% irb
>> require 'hola'
=> true
>> Hola.hi
Hello world!
```

B.1.3.2 Requiring More Files

B.1.3.3 Adding An Executable

B.1.3.4 Writing Tests

B.1.3.5 Documenting Your Code

Appendix C Resources

Here are some additional resources that I have found helpful.

C.1 Ruby Association

Ruby Association

Ruby Association is a organization dovoted to advancing Ruby programming language.

Mission Statement

The Ruby Association is a non-profit organization devoted to the advancement of the Ruby programming language. Our goal is to strengthen relationships between Ruby-related projects, communities, and businesses and to work towards solving the problems faced by Ruby users.

C.1.1 Ruby Association Certified Ruby Programmer Examinations

The Ruby Association Certified Ruby Programmer examinations are intended for engineers who design, develop, and/or operate Ruby-based systems, consultants who make Ruby-based system proposals, and instructors who teach Ruby. Those who are certified are recognized for their skills as Ruby engineers and as having high levels of Ruby-based system development capabilities. Those who pass the examination are certified by the Ruby Association as a Ruby Association Certified Ruby Programmer.

C.1.1.1 Ruby Association Certified Ruby Programmer Silver version 2.1

This is a basic skill-level certification of the knowledge on the background, grammar, classes, objects, and standard libraries of Ruby. The logo below identifies people who are certified.

Scope

- SYNTAX
  - Comment
  - Literal (e.g., numbers, booleans, strings, characters, arrays, hashes)
  - Variables, constants, and scope
  - Operators
  - Conditional branching
  - Loops
  - Exception handling
  - Method calls
  - Blocks
  - Method definition
  - Class definition
  - Module definition

    - Multilingualization
- BUILT-IN LIBRARIES
    - Well-used built-in classes and modules (e.g., Object, Numerical classes, String, Array, Hash, Kernel, Enumerable, Comparable)
- OBJECT ORIENTATION
    - Polymorphism
    - Inheritance
    - mix-in

C.1.1.2 Ruby Association Certified Ruby Programmer Gold version 2.1

This is a certification of a deeper understanding of the topics covered under Silver certification (syntax, object oriented programming, embedded libraries, operating environments, etc.) in addition to knowledge of the standard Ruby libraries and knowledge related to classes and objects required for Ruby application design.

Scope

- EXECUTION ENVIRONMENT
    - Command line options
    - Pre-defined variables and constants
- SYNTAX
    - Variables and constants Operators
    - Blocks
    - Exceptions
    - Non local exit
    - Keyword arguments
    - lambda
- OBJECT-ORIENTED PROGRAMING
    - Details of Method
    - Access control
    - Details of Class
    - Class inheritance
    - Details of modules
    - Module#prepend
    - Refinements
- BUILT-IN LIBRARIES
    - Well-used built-in classes and modules e.g., Object, Module, Kernel, Enumerable, Comparable
    - Numeric
    - Regular expression
    - Proc

    - Thread/Fiber
- STANDARD LIBRARY
    - Well-used standard librariese.g., socket, date, stringio

C.1.1.3 Ruby Association Certified Ruby Programmer Platinum

Under development

C.1.2 Study Materials

- Ruby Association official prep test
- New prep test for the Silver exam
- The Ruby Programming Language
- Programming Ruby

C.2 Programming Ruby by Hulan

Programming Ruby (See item *ProgrammingRuby* in "Bibliography", page 239.)

By Marek Hulan, Marek Jelen, Ivan Necas, Version 0.2 September 25, 2017.

C.2.1 Ruby Basics

C.2.1.1 Installation - Running - About

Installation on Mac OS

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

```
$ brew install ruby
```

In case there is a need to have multiple installations `rbenv` (see Section 2.1.3.2 "`rbenv`", page 4) is a good tool to use. Another popular tool is called `rvm` (see Section 2.1.3.3 "RVM ("Ruby Version Manager")", page 4, but nowadays they do too much magic and we recommend using `rbenv` instead.

Running Ruby

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

```
$ ruby script.rb
```

Interactive Ruby

Ruby comes with an interactive interpreter called `irb` (see Section 2.4.5.1 "Interactive Ruby", page 33), which allows you to enter commands and see results instantly.

```
$ irb
```

About Ruby

Interpreted

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

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

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

Everything Is An Expression
Everything command in Ruby has a return value.

Atomic memory management

MRI *MRI* ("Matz's Ruby Interpreter") (aka `CRuby`) is the Ruby reference implementation, with several other implementations like `JRuby`, `Rubinius`, `IronRuby` or `MagLev`. As well, there is ISO standard implemented in `mruby`.

YARV Starting with Ruby 1.9, and continuing with Ruby 2.x and above, the official Ruby interpreter has been YARV ("Yet Another Ruby VM"). *YARV* is a bytecode interpreter that was developed for the Ruby programming language by Koichi Sasada. The goal of the project was to greatly reduce the execution time of Ruby programs. Since YARV has become the official Ruby interpreter for Ruby 1.9, it is also named KRI (Koichi's Ruby Interpreter), in the same vein as the original Ruby MRI, named for Ruby's creator Yukihiro Matsumoto.

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

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

What is written in Ruby?

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

C.2.1.2 Language Conventions

Class Names

Class names are in "CamelCase"

File Names

File names reflect class names in "snake_case" format

Method Names

method names are also "snake_case"

Constants

constants are in UPPER_SNAKE_CASE

Indentation

indent by 2 spaces

Boolean Values

methods returning boolean values end with `?` (question mark)

Mutation

methods mutating state end with `!` (exclamation mark)

C.2.1.3 A Little About Data Types

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

Strings

String are create by quoting the characters.

Numbers

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

Empty Value `nil`

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

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

Boolean

There is either `true` or `false`.

Arrays

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

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

Hashes

A *Hash* is a structure that maps keys to a values.

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

Hash Styles

There are two approaches how to write the mapping, either *rocket* style: '`key => value`' or *json* style: '`key: value`'.

`Rocket` Style Preferred

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

Symbols

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

C.2.1.4 Methods and Variables

Methods

Methods are called using the "dot" operator.

Operators are actually methods.

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

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

Variables

Global *Global variables* are prefixed with *$*.

Class Classes have *class variables* prefixed with `@@`.

You should prefer `@instance_variables` in class-level methods, as they have more predicatable behavior.

Instance Objects have *instance variables* prefixed with `@`.

Local *Local variables* are simple identifiers with no prefix.

Constant *Constants* are in all UPPER_CASE.

C.2.1.5 Conditions

Everything is `true` except `false` (and `nil`).

`if` and `unless` (`if`'s negative)

Modifier Statements

Ruby has inline method of using conditionals called *modifier statements*.

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

Ternary Operator

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

Case Statement

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

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

`case` statement can also check on a variable's `class`.

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

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

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

C.2.1.6 A Few Logical Operators

Logical Expressions

and `&& and`

or `|| or`

not `! not`

Comparison Operators

equal `==`

not equal `!=`

less than `<`

greater than `>`

less than or equal `<=`

greater than or equal
`>=`

regular expression match
`=~`

C.2.1.7 Regular Expressions

Regular expression literals are enclosed between `/.../`:
'`string =~ /(.)*:(.)+/ # sets $1 a $2`'.

String has a `match` method:
'`data = string.match(/^(.):(\d+)$/) # sets data[1] and data[2]`'.

C.2.1.8 Loops

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

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

# do loop at least once
begin
  a += 1
end while a < b

a += 1 while a < b # inline loop
```

`until` repeats as long as a condition is false

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

C.2.1.9 Methods

Method Return Value

Methods always return a value. If a method does not explicitly return using the `return` keyword, then the return value is the value of the last expression.

Method Arguments

```
def mth1(a, b=1) # argument default value
end

def meth2(*args) # accepts multiple arguments in Array args
end

def meth3(a, b, *args) # must have at least 2 arguments
end
```

C.2.1.10 Using Code From Other Files

`require` Loads code from another file. Ruby keeps track of required files (in the `$"` global variable) and skips loading files that would be loaded 2nd time.

Files are looked up using Rubys *load path*, which is represented using an array in `$LOAD_PATH` and `$:`.

`load` loads code from another file, but does not keep track of loaded files.

Allowed Extensions

The following file extensions can be omitted:

'`rb, so, o, dll, bundle, jar`'

C.2.1.11 Blocks

Blocks are not executed when defined, but have to be called through the `call` method (though the calling of the `call` method is most of the time hidden).

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

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

# alternative syntax
arr.each { |el| puts el }
```

Block Scope

Blocks have access to their enclosing environment, and create their own scope internally.

Yield Statement

Any method can accept a block and call it using `yield`.

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

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

Block Arguments Using Prefix `&`

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

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

C.2.1.12 Objects

Everything is an object. Even a class is an instance of `Class`. Objects can have methods (`def`) and instance variables (`@`).

Attributes

To allow instance variables to be accessible from outside their defining class, define them using *attributes*.

```
class Hello
  attr_reader :one
  attr_writer :two
  attr_accessor :three
end
```

C.2.1.13 Inheritance

Ruby has single *object inheritance*. All methods, including constructors, are inherited. Methods can be *overridden* in children. `super` keyword is used to call the overridden method.

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

puts A.new.a # => hello

class B < A
end

puts B.new.a # => hello

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

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

Class Variables

Class variables should not be used "because of some pitfalls in their inheritance."

C.2.1.14 Modules

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

```
class A
```

```
  class B
  end
end

module Some
  class Thing
  end
end
```

Mixins

Modules are used as *mixins* to provide a form of multiple inheritance. When module is included in a class, all methods defined in that module are available in the class as instance methods.

```
module Helper
  def something
  end
end

class An
  include Helper
end

A.new.something
```

When a module is used with `extend`, the methods are included as class methods.

```
module Helper
  def something
  end
end

class A
  extend Helper
end

A.something
```

C.2.1.15 Method Access

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

```
class A
  def public_method
  end

  protected

  def protected_method
  end
```

```
  private

  def private_method
  end
end
```

C.2.1.16 Duck Typing

Duck typing is coding by behavior rather than identify.

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

If an animal "quacks", then it will be shot like a duck.

C.2.1.17 Exceptions

Exceptions represent a special state in the execution of a program. When an `exception` is raised, it will "bubble up" (rise through) the stack until it is "caught".

`raise` And `rescue` Exceptions

Exceptions are explicitly raised using the `raise` keyword: '`raise "This is not expected"`'.

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

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

Inherit From `StandardError`

Dont inherit directly from an `Exception` class; instead, use `StandardError`. The direct descendants of `Exception` are usually exceptions one doesn't want to rescue from, such as `SystemExit` or `NoMemoryError`.

C.2.2 Advanced Ruby

C.2.2.1 Return Values

In Ruby everything is an expression — it executes and returns some value, the result of its execution. For example:

```
5 + 2        # ⇒ 7
```

```
"hello"      # ⇒ "hello"

"a" if true  # ⇒ "a"
"a" if false # ⇒ nil

class A; end # ⇒ ???
```

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

```
class A; end # ⇒ nil
```

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

```
class A; 1; end          # ⇒ 1
class A; 1; "hello"; end # ⇒ "hello"
class A; self; end       # ⇒ A
```

It was said that "the return value of a method is the result of its last expression." Its obvious that in this example its very similar. It can be generalized for all structures, even though sometimes it may not be obvious on first sight:

The return value is the result of the last expression.

C.2.2.2 Context

Context: Current Context

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

```
self.class # ⇒ Object

class B
  self
end
# ⇒ Class

class A
  def call
    self
  end
end

A.new.call # ⇒ #<A:some number>
```

C.2.2.3 Class

Open Classes: Monkey Patching

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

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

class Clazz
  def call
    "B"
  end
end

Clazz.new.call # ⇒ "B"
```

What Is A Class?

Lets start with a definition:

> Classes are instances of class `Class`.

Everything in Ruby is an object . . . even a class.

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

B = Class.new
def B.call
  "called"
end

A.call      # ⇒ "called"
B.call      # ⇒ "called"

A.object_id # ⇒ [some number]
B.object_id # ⇒ [some number]

A.class     # ⇒ Class
B.class     # ⇒ Class
```

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

Inheritance

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

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

class B < A
end

C = Class.new(B)

B.new.call # ⇒ "called"
C.new.call # ⇒ "called"
```

Mixins

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

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

class A
  include Methods
end

A.new.call # ⇒ "called"
```

Class Introspections

Ruby allows many introspections on classes and many other objects.

`name`

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

```
Array.name    # ⇒ "Array"

[].class.name # ⇒ "Array"
```

`methods`

Its possible to list methods of an object:

```
class A
```

```
  def call
  end
end

A.new.methods # ⇒ array of methods
```

C.2.2.4 Advanced Methods

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

Extracting Methods

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

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

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

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

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

Checking Method Existence

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

Duck Typing

> Do not care what the object is; only care whether it behaves as expected.

This technique is called "Duck typing."

```
class A
  def call
  end
end

a = A.new
a.respond_to?(:call) # ⇒ true
a.respond_to?(:wtf)  # ⇒ false
```

Dynamic Method Calling

Lets define a class with a method, create an instance and call the method.

```
class A
```

```
  def call
  end
end

A.new.call
```

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

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

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

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

Access Modifiers

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

```
class A
  def public_method
  end

  protected

  def protected_method
  end

  private

  def private_method
  end

end

a = A.new
a.public_method           # ⇒ nil
a.protected_method        # ⇒ NoMethodError: protected method ‘protected_method’ call
a.private_method          # ⇒ NoMethodError: private method ‘private_method’ called .
a.send(:public_method)    # ⇒ nil
a.send(:protected_method) # ⇒ nil
a.send(:private_method)   # ⇒ nil
```

Defining methods programmatically

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

However, `Class.define_method` is private. To get around this obstacle, its possible to use the `send` method and bypass the access modifier.

```
class A
end

a = A.new

logic = Proc.new do
  "data"
end

A.send(:define_method, :some_method_name, logic)
a.some_method_name # ⇒ "data"
```

Missing Methods

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

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

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

C.2.2.5 Advanced Objects

Objects complement classes:

> Objects define state while classes define behavior.

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

Creating A New Object

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

```
class Dog
end

dog = Dog.new
```

Defining Methods

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

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

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

`def` defines methods into the nearest class.

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

```
var = class A; self; end

var.class # ⇒ Class
var.name  # ⇒ "A"
```

OK, so the the current context is a Class and thus it is obvious that the nearest class is this class. Now lets try to define a class method:

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

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

Eigenclass

To understand how Ruby works, we have to understand what *eigenclasses* are. Lets start with simple definition:

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

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

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

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

Back to the example we were talking about:

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

To see it more clearly we can rewrite this example identically as:

```
class A
end

def A.call
  "string"
end
```

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

```
class A
end

scope = class A
  self
end

A == scope # ⇒ true
```

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

There are no class methods in Ruby

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

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

```
eigenclass = class << some_object
  self
end

eigenclass = some_object.singleton_class
```

Now that we can access eigenclasses, lets see how we could define "class methods" (instance methods in the eigenclass).

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

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

```
end

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

All those examples are identical.

Method Lookups

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

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

and for objects

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

Though in reality it is a bit more complex.

Eigenclasses are not visible as classes of objects.

```
o1 = Object.new

def o1.meth
  "string"
end

o1.meth  # ⇒ "string"
o1.class # ⇒ Object

o2 = Object.new

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

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

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

C.2.3 Ruby Testing

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

C.2.3.1 Testing Frameworks

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

C.2.3.2 Minitest

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

`Testunit` Syntax

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

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

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

`Minitest::Test`

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

There are a few method names with special meaning. In the example you can see a method with th name `setup`. This method gets automatically executed before every testing method. Similarly theres a `teardown` method that gets executed after each testing method. Its usually used for cleaning up any mess the testing method created.

Assertions

`skip()` One testing method can contain more than one assertion. The first assertion failure stops the method being run and marks the test as a failure (F). If method raises an exception the result of the test is marked as an error (E). If all assertions defined in the method pass, the test succeeds (.). If you plan to implement the test later you can skip the test by calling `skip("Sorry, Im lazy")`.

The simplest assertion is one testing a boolean value:

```
assert @something
```

`assert refute`

This will succeed if `@something` is considered `true`, but fail otherwise. The negative form is `refute`, e.g. the following would pass: '`refute false`'.

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

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

```
string'
```

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

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

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

```
assert arg
          arg is true

refute arg
          arg is false

assert_equal
          expected, actual

assert_includes
          collection, object

assert_kind_of
          class, object

assert_nil
          object

assert_match
          expected, actual

assert_raises
          exception_class, &block
```

Table C.1: Table of Assertion Helpers

Spec Syntax

`Spec` Syntax

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

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

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

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

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

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

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

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

Output Of Test Run

```
Run options: --seed 25127

# Running tests:

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

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

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

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

Running Multiple Test Files

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

```
require 'rake/testtask'

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

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

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

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

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

Test Coverage

To get a good overview of what needs test coverage its useful to setup code coverage check. A `simplecov` gem can generate an HTML report. Just put the following on top of your `test_helper.rb`:

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

you can also define a minimum coverage in percents:

```
SimpleCov.minimum_coverage 95
```

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

Stubbing

Sometimes we dont want to call the entire method chain when we test — just a single method behavior. This applies especially in unit testing where we test just small piece of code. Since Ruby is a dynamic language, its easy to cut off some methods. This is called *stubbing* (leaving stubs).

Lets look at the following example:

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

The test covering this should call the method `measure` and verify it returns a reasonable temperature. We dont want our test to print anything to `STDOUT`. We can stub out `puts` method easily like this:

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

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

Mocking

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

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

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

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

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

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

Stubbing Network Calls

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

`webmock` Gem

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

```
require 'webmock/minitest'
stub_request(:get, 'www.example.com')
First is the
Net::HTTP.get('www.example.com', '/') # this will succeed
```

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

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

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

`vcr`

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

```
require 'vcr'

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

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

C.2.3.3 Testing Web Applications

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

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

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

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

Capybara.app = MyApp
Capybara.default_driver = :rack_test

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

  def test_index
    visit '/'
```

```
    click_link 'login'
    fill_in('Login', with: 'Marek')
    fill_in('Password', with: 'secret')
    click_button('Submit')

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

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

C.2.3.4 Cucumber

We can use the `Cucumber` framework for testing using the BDD aproach. It allows us to write the behavior specification in a natural language style first and then convert it to tests step-by-step. Imagine youd describe a feature like this:

```
Feature: logout of logged in user

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

Its a valid `cucumber` test (aka feature) which only needs implementing those steps, using `capybara` for example.

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

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

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

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

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

Appendix D Utility Programs

Here are some utility programs that I either found or created.

D.1 Ruby Eval Utility

See "On Simple Examples", page 58,

`eval.rb`

NOTE: This program's original name is `eval.rb`. However, there is a name conflict with `eval`. In order to allow this program to be run as an executable while sitting in the `bin` directory, I need to change its name to soemthing other than `eval`. Of course, `eval.rb` would work, but I would like a simple name without an extension. Therefore, I am using `rbeval` as a compromise. This is handled by `@post_create` when creating an executable in the `bin` directory. The original file will be moved into `src` with its original `eval.rb` intact.

{`eval.rb`} ≡

```
#!/usr/bin/env 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 216.
<*eval—EvalWrapper–Indentation Deltas*>	See “`eval.rb` Indentation Deltas Code”, page 216.
<*eval—Main–Get Line*>	See “`eval.rb` Main Get Line Code”, page 217.
<*eval—Main–Process Line*>	See “`eval.rb` Main Process Line Code”, page 217.

D.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 “Ruby Eval Utility”, page 215.

D.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 "Ruby Eval Utility", page 215.

D.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 "Ruby Eval Utility", page 215.

D.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 "Ruby Eval Utility", page 215.
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 218.

<*eval—Main–Process Line-Is Line*> See "`eval.rb` If Is Line Code", page 218.

D.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 217.

D.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 217.

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 218.
<*eval—Main–Process Line_Worth Evaluating?*>	*Line-Is*	See "`eval.rb` If Is Line Code", page 219.

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 218.

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 218.

D.1.5 eval.rb Post Create

The following line is executed after `jrtangle` runs.

```
@post_create eval.rb chmod +x eval.rb && mv eval.rb src && ln src/eval.rb bin/rbeval
```

D.2 API Utility

The purpose of this little utility is to assist with the construction of the API tables; that is, it will help insert `@item` labels, and perhaps assist with constructing URL's if I am lucky. I am writing it using the `GAWK` programming language to refresh my knowledge of `AWK` (and get to know `GAWK`'s extra features).

{`apiutil.awk`} ≡

```
#! /usr/bin/env gawk -f
```

<*apiutil—BEGIN Block*>
<*apiutil—BEGINFILE Block*>
<*apiutil—MAIN Block*>
<*apiutil—ENDFILE Block*>
<*apiutil—END Block*>

The following table lists called chunk definition points.

Chunk name	**First definition point**
<*apiutil—BEGIN Block*>	See "`apiutil.awk` BEGIN Block", page 220.
<*apiutil—BEGINFILE Block*>	See "`apiutil.awk` BEGINFILE BLOCK", page 220.
<*apiutil—END Block*>	See "`apiutil.awk` END Block", page 226.
<*apiutil—ENDFILE Block*>	See "`apiutil.awk` ENDFILE Block", page 226.
<*apiutil—MAIN Block*>	See "`apiutil.awk` MAIN Block", page 220.

D.2.1 `apiutil.awk` BEGIN Block

<*apiutil—BEGIN Block*> ≡

```
BEGIN {
  <apiutil—BEGIN–Variable Defns>
  <apiutil—BEGIN–Ord Function Init>
}
```

This chunk is called by {`apiutil.awk`}; see its first definition at "API Utility", page 219.
The following table lists called chunk definition points.

Chunk name	**First definition point**
<*apiutil—BEGIN–Ord Function Init*>	See "`apiutil` Ord Function", page 226.
<*apiutil—BEGIN–Variable Defns*>	See "`apiutil.awk` MAIN Block", page 221.

D.2.2 `apiutil.awk` BEGINFILE BLOCK

This will allow this filename to be changed a little more easily.

<*apiutil—BEGINFILE Block*> ≡

```
BEGINFILE {
  newfile = FILENAME".new"
}
```

This chunk is called by {`apiutil.awk`}; see its first definition at "API Utility", page 219.

D.2.3 `apiutil.awk` MAIN Block

<*apiutil—MAIN Block*> ≡

```
###############
# MAIN BLOCK #
###############

# pass through whatever is not in a target table
started == 0 { print > newfile }
```

<*apiutil—MAIN Block–Main Loop*>

```
########################
# FUNCTION DEFINITIONS #
########################
```

<*apiutil—MAIN Block–Ord Function Defn*>
<*apiutil—MAIN Block–Convert Symbols Function Defn*>

```
##################
# END MAIN BLOCK #
# ################
```

This chunk is called by {`apiutil.awk`}; see its first definition at "API Utility", page 219.
The following table lists called chunk definition points.

Chunk name	**First definition point**
<*apiutil—MAIN Block–Convert Symbols Function Defn*>	See "`convertsymbols()` Function Definition", page 227.
<*apiutil—MAIN Block–Main Loop*>	See "`apiutil.awk` MAIN Block", page 221.
<*apiutil—MAIN Block–Ord Function Defn*>	See "`apiutil` Ord Function", page 227.

Main Loop Variable Definitions

I will define the main loop variables first. The variables `start` and `end` hold regular expression strings that target the two tables I want to process. The variable `started` flags whether the code is inside a table (1) or not (0).

<*apiutil—BEGIN–Variable Defns*> ≡

```
start = "^@float Table,table:api-.*$"
end   = "^@end table$"
started = 0
```

This chunk is also defined in "`apiutil.awk` MAIN Block", page 224, "`apiutil.awk` MAIN Block", page 224, and "`apiutil.awk` MAIN Block", page 224.

This chunk is called by <*apiutil—BEGIN Block*>; see its first definition at "`apiutil.awk` BEGIN Block", page 220.

Main Outer Loop Definitions

The `MAIN` block does two things: passes over unchanged whatever is not inside a target table, and processes the contents of a target table. It uses the flag `started` to pass through the beginning and ending lines basically unchanged as well[1].

<*apiutil—MAIN Block–Main Loop*> ≡

```
# process whatever is in between the 'start' and 'end' of a table
```

[1] In fact one line is added at the end of the table to account for better style

```
$0 ~ start, $0 ~ end {

  # Upon first entering a table, set the 'started' flag true (1)
  if (started == 0) {
    started = 1
    next
  }

  <apiutil—MAIN Block–Main Loop-Inside>
}
```

This chunk is called by <*apiutil—MAIN Block*>; see its first definition at "`apiutil.awk` MAIN Block", page 220.

The called chunk <*apiutil—MAIN Block–Main Loop-Inside*> is first defined at "`apiutil.awk` MAIN Block", page 222.

Preliminary and Post Processing of the Inside of a Table

Once inside the main loop, do some preliminary processing and checking. Make sure `@` signs are properly escaped, and ignore the `@table` line. Also check for the end of the table, add a newline, and reset the `started` flag.

<*apiutil—MAIN Block–Main Loop-Inside*> ≡

```
# ignore the @table line
if ($0 ~ /^@table/) {
  print $0 > newfile
  next
}

# at the end of a table, add an empty line for better style
# and turn off the 'started' flag
if ($0 ~ end) {
  print "\n"$0 > newfile
  started = 0
  next
}

# make sure any special symbols are properly escaped
gsub(/@/, /@@/)
```

This chunk is also defined in "`apiutil.awk` MAIN Block", page 223.

This chunk is called by <*apiutil—MAIN Block–Main Loop*>; see its first definition at "`apiutil.awk` MAIN Block", page 221.

Main Processing Loop Definition

The real work is done on lines that are in the middle of the tables. There is first a regular expression check to make sure something that is not a `Texinfo` command (which begin with `@`) is on the line because some lines are empty and are just ignored;

<*apiutil—MAIN Block–Main Loop-Inside*> +≡

```
# process lines with content
if ($0 ~ /^[CM]?[[:graph:]]+.*$/) {

  <apiutil—MAIN Block–Main Loop-Inside_Processing>
  # ignore empty lines
} else {
  print > newfile
}
```

This chunk is also defined in "apiutil.awk MAIN Block", page 222.

This chunk is called by <*apiutil—MAIN Block–Main Loop*>; see its first definition at "apiutil.awk MAIN Block", page 221.

The called chunk <*apiutil—MAIN Block–Main Loop-Inside_Processing*> is first defined at "apiutil.awk MAIN Block", page 223.

Inside the Main Processing Loop

After it finds a non-empty, non-command line, GAWK parses it into the array `arr` using the function `match`. This regexp is fairly busy. There are many parentheses because several parts of the regexp are optional, requiring an extra set around the optional parts, and extra parentheses are required to strip away a parenthesized word (ironically). The optional sections are marked by question marks (`?`s).

The first part of each line contains a marker indicating what kind of element the thing is. In the `Classes` table, the marker is either a `C` or an `M`. In the `Methods` table, the marker is either a pair of colons (`::`) or a hash (`#`).

Explanation of the main regexp

The main entry from either table is obtained by grabbing everything that is not a right parenthesis (`(`) or a space. In the `Class` table, this is everything on the line. In the `Method` table, this is everything up until the parenthesized class name in which the method is defined. Likewise, this parenthesized class is obtained by grabbing everything up until the closing parenthesis.

Finally, each parsed parenthesized regexp is inserted into the array `arr` and available for use after the parsing.

1. `arr[1]`: Identification of either `class` or `method`; it will contain one of *`C`*, *`M`*, *`::`*, or *`#`*.
2. `arr[2]`: Main entry; it will contain the name of a class (for the `Class` table, or the name of a method (for the `Method` table).
3. `arr[3]`: ignored parenthesized class (optional)
4. `arr[4]`: a method's `class` (optional); it will contain the name of the class within which a method is defined.

<*apiutil—MAIN Block–Main Loop-Inside_Processing*> ≡

```
cm = match($0, /([CM]|[:#]+)?([^( ]+)(\s\(([^)]+)\))?/, arr)

# for debugging
```

```
# print "1-"(arr[1])"-2-"(arr[2])"-3-"(arr[3])"-4-"(arr[4])"|"
```

This chunk is also defined in "apiutil.awk MAIN Block", page 225, "apiutil.awk MAIN Block", page 225, and "apiutil.awk MAIN Block", page 226.

This chunk is called by <*apiutil—MAIN Block–Main Loop-Inside*>; see its first definition at "apiutil.awk MAIN Block", page 222.

Variable Definitions

When parsing the lines of text obtained from copying the library page, the initial letters and symbols can be translated into meaningful words using the hash `classmethod`.

<*apiutil—BEGIN–Variable Defns*> +≡

```
classmethod["C"]  = "Class"
classmethod["M"]  = "Module"
classmethod["::"] = "Class Method"
classmethod["#"]  = "Instance Method"
```

This chunk is also defined in "apiutil.awk MAIN Block", page 221, "apiutil.awk MAIN Block", page 224, and "apiutil.awk MAIN Block", page 224.

This chunk is called by <*apiutil—BEGIN Block*>; see its first definition at "apiutil.awk BEGIN Block", page 220.

Additionally, a method's designator can be used to help form URL's pointing to the method definition.

<*apiutil—BEGIN–Variable Defns*> +≡

```
methid["::"] = "-c-"
methid["#"]  = "-i-"
```

This chunk is also defined in "apiutil.awk MAIN Block", page 221, "apiutil.awk MAIN Block", page 224, and "apiutil.awk MAIN Block", page 224.

This chunk is called by <*apiutil—BEGIN Block*>; see its first definition at "apiutil.awk BEGIN Block", page 220.

Stitching Together the Parts

Once the parsing has been completed, the parts are stitched together and printed with appropriate commands surrounding them, i.e., `@item` or `@code`, etc.

The variable `itemurl` will be used to stitch together the URL that will be the main `@item` content; it begins with the base url as defined in the BEGIN block:

<*apiutil—BEGIN–Variable Defns*> +≡

```
baseurl = "http://ruby-doc.org/core-2.5.1/"
```

This chunk is also defined in "apiutil.awk MAIN Block", page 221, "apiutil.awk MAIN Block", page 224, and "apiutil.awk MAIN Block", page 224.

This chunk is called by <*apiutil—BEGIN Block*>; see its first definition at "apiutil.awk BEGIN Block", page 220.

`baseurl` starts the `itemurl` definition:

<*apiutil—MAIN Block–Main Loop-Inside_Processing*> +≡

```
itemurl = (baseurl)
```

This chunk is also defined in "`apiutil.awk` MAIN Block", page 223, "`apiutil.awk` MAIN Block", page 225, and "`apiutil.awk` MAIN Block", page 226.

This chunk is called by <*apiutil—MAIN Block–Main Loop-Inside*>; see its first definition at "`apiutil.awk` MAIN Block", page 222.

Processing the `arr` Array

The additional parts of the array `arr` will be added to `itemurl` to create a complete and linkable URL into Ruby's core library pages.

The code now continues processing the `arr` array parts. If `arr[4])` exists, then the code is currently processing the `Method` table, and since the parts are different than the `Class` table, it must differentiate between them.

`arr[4]` will be empty for a `Class` table element, and so it can be reformatted somewhat for a `Method` table element by placing its contents inside a `@code` format.

The `Method` links require most symbols to be hexadecimal equivalents of their ASCII code number. Therefore, I made a function called `convertsymbols()` that will iterate over a method name and convert any symbols (with the exception of the underscore) into hexadecimal ASCII, separated by dashes. When this name is inserted into the URL, it links directly to the proper method inside the class documentation.

<*apiutil—MAIN Block–Main Loop-Inside_Processing*> +≡

```
if (length(arr[4]) > 0) {

  #process the Method table
  method = arr[2]
  class  = arr[4]
  gsub(/::/, "/", class)
  methwsymbols = convertsymbols(method)

  itemurl = "@item @url{"(itemurl)(class)".html#method"(methid[arr[1]])(methwsymbols)"
  detail  = "@code{"(class)(arr[1])(method)"} ("(classmethod[arr[1]])")"

} else {
  # process the Class table
  class = arr[2]

  itemurl = "@item @url{"(itemurl)(class)".html,"(arr[2])"}"
  detail  = (classmethod[arr[1]])
}
```

This chunk is also defined in "`apiutil.awk` MAIN Block", page 223, "`apiutil.awk` MAIN Block", page 225, and "`apiutil.awk` MAIN Block", page 226.

This chunk is called by <*apiutil—MAIN Block–Main Loop-Inside*>; see its first definition at "`apiutil.awk` MAIN Block", page 222.

Printing the Table Items

The code is printing a `@table` element, and it contains both an `@item` column and a detail column.

<*apiutil—MAIN Block–Main Loop-Inside_Processing*> +≡

```
print itemurl > newfile
print detail  > newfile
```

This chunk is also defined in "`apiutil.awk` MAIN Block", page 223, "`apiutil.awk` MAIN Block", page 225, and "`apiutil.awk` MAIN Block", page 225.

This chunk is called by <*apiutil—MAIN Block–Main Loop-Inside*>; see its first definition at "`apiutil.awk` MAIN Block", page 222.

D.2.4 `apiutil.awk` ENDFILE Block

<*apiutil—ENDFILE Block*> ≡

```
ENDFILE {
  system("mv -v "FILENAME" "FILENAME".bak && mv -vi "FILENAME".new "FILENAME)
}
```

This chunk is called by {`apiutil.awk`}; see its first definition at "API Utility", page 219.

D.2.5 `apiutil.awk` END Block

<*apiutil—END Block*> ≡

```
END {
  print "All done"
}
```

This chunk is called by {`apiutil.awk`}; see its first definition at "API Utility", page 219.

D.2.6 `apiutil` Ord Function

I need to turn symbols (like `=` and `+`) into their hexadecimal numbers inside `Method` URL links. I found this `ord` function in `GAWK`'s documentation. See Section "Ordinal Functions" in *GAWK*. It consists of an initialization segment, in which a hash of symbols and their corresponding ASCII codes is assembled, and a function that, given either an `ord` number or a `char` symbol, returns the opposite. I will put the initialization code into the `BEGIN` segment, and the function definition into the `MAIN` block.

The `ord` Function Initialization Segments

The `_ord_init()` initialization function is called from the BEGIN block, but is defined at the bottom of the MAIN block along with the other `ord` functions.

<*apiutil—BEGIN–Ord Function Init*> ≡

```
# initialize the _ord_ and _chr_ function's ASCII symbol table
_ord_init()
```

This chunk is called by <*apiutil—BEGIN Block*>; see its first definition at "`apiutil.awk` BEGIN Block", page 220.

<*apiutil—MAIN Block–Ord Function Defn*> ≡

```
# initialize the ASCII symbol table
#
function _ord_init(       low, high, i, t) {
  low = 0
  high = 127
  for (i = low; i <= high; i++) {
    t = sprintf("%c", i)
    _ord_[t] = i
  }
}
```

This chunk is also defined in "`apiutil` Ord Function", page 227.

This chunk is called by <*apiutil—MAIN Block*>; see its first definition at "`apiutil.awk` MAIN Block", page 220.

The `ord()` and `chr()` Function Definitions

<*apiutil—MAIN Block–Ord Function Defn*> +≡

```
# the 'ord()' function: given a symbol, return its ASCII number
#
function ord(str,       c) {
  c = substr(str, 1, 1)
  return _ord_[c]
}

# the 'chr()' function: given an ASCII number, return its symbol
#
function chr(c) {
  return sprintf("%c", c + 0)
}
```

This chunk is also defined in "`apiutil` Ord Function", page 227.

This chunk is called by <*apiutil—MAIN Block*>; see its first definition at "`apiutil.awk` MAIN Block", page 220.

D.2.7 `convertsymbols()` Function Definition

This function, `convertsymbols()`, takes a method name and converts all symbols in the following ranges into their hexadecimal equivalents:

- ASCII 0x20 (`SP`) and ASCII 0x2f (`/`)
- ASCII 0x3a (`:`) and ASCII 0x40 (`@`)
- ASCII 0x5b (`[`) and ASCII 0x5e (`^`)
- ASCII 0x7b (`{`) and ASCII 0x7e (`~`)

<*apiutil—MAIN Block–Convert Symbols Function Defn*> ≡

```
# the 'convertsymbol()' function: given a method name, convert symbols
```

```
# into hexadecimal separated by dashes
#
function convertsymbols(meth,\
  converted, low1, high1, low2, high2, low3, high3, low4, high4, c, f, i, o) {

  # ASCII ranges to look for
  low1  = 0x20 # SP
  high1 = 0x2f # /
  low2  = 0x3a # :
  high2 = 0x40 # @
  low3  = 0x5b # [
  high3 = 0x5e # ^ (leave _ alone)
  low4  = 0x7b # {
  high4 = 0x7e # ~

  # this flag places dashes between symbols
  dash = 0
  f[1] = "-"

  # this will hold the converted name
  converted = ""

  # iterate over the characters in 'meth', converting the symbols
  for (i = 1; i <= length(meth); i++) {
    c = substr(meth, i, 1)

    # this is a decimal number internally, but is converted to hexadecimal
    # by the "%x" format string
    o = ord(c)

    if ( (o >= low1 && o <= high1) ||
         (o >= low2 && o <= high2) ||
         (o >= low3 && o <= high3) ||
         (o >= low4 && o <= high4) ) {

      # do not place a dash if a symbol is the first character
      converted = (converted)(sprintf("%s%X", f[dash], o))

    } else {
      converted = (converted)(c)
    }

    if (dash == 0) { dash = 1 }
  }

  return converted
}
```

This chunk is called by <*apiutil—MAIN Block*>; see its first definition at "`apiutil.awk` MAIN Block", page 220.

D.2.8 apiutil Makefile Target

Add a target for running `apiutil.awk` in the Makefile. Note that the directory `./bin` is created in the `@initial_setup` environment (see Section E.1 "Initial Setup", page 230) and is made executable by a `@post_create` command (see Section E.2 "Post Create", page 230), both of which were added by the `TexiwebJR` program.

<*Makefile—Utility Targets*> ≡

```
# apiutil.awk
#############
.PHONY : apiutil
apiutil :
    bin/apiutil.awk Ruby2_5.twjr
```

This chunk is also defined in "Utility Targets", page 233.

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 231.

Appendix E Initial Setup and Post Create

TexiwebJr has a couple of new utility commands for working with files:

- `@initial_setup`
- `@post_create`

E.1 Initial Setup

TexiwebJR added a new command `@initial_setup` that executes some commands in the shell prior to `jrtangle` or `jrweave` processing a file. This can be used to create directories into which the command `@post_create` can move specified files (see Section E.2 "Post Create", page 230).

Since these commands do not show up in a woven document, I have placed them into a little `@example` environment here:

```
@initial_setup
mkdir -p src
mkdir -p bin
@end initial_setup
```

Set up some directories into which files can be moved, and add a little shell script to add `./bin` to `PATH` (but it can only be run from the command line as *`source setpath.sh`*).

{`setpath.sh`} ≡

```
#! /usr/bin/env bash
# setpath.sh USAGE 'source setpath.sh'

export PATH=./bin:$PATH
```

E.2 Post Create

Make `apiutil.awk` executable and move it into `bin`. It can be run either by executing the `make` target '`apiutil`' (as '`make apiutil`') or by running the command from the `./bin` directory. Note that there is a little shell script that will add `./bin` to the `path` environment variable, which must be run "by hand" as '`source setpath.sh`'.

```
@post_create apiutil.awk chmod -v +x apiutil.awk && mv apiutil.awk src && ln -vf src/apiut
```

Appendix F The Makefile

{Makefile} ≡

```
# MAKEFILE FILE CHUNKS
######################

<Makefile—Variable Definitions>

<Makefile—Default Target>

<Makefile—TWJR Targets>

<Makefile—Utility Targets>

<Makefile—Clean Targets>
```

The following table lists called chunk definition points.

Chunk name	First definition point
<*Makefile—Clean Targets*>	See "Clean Targets", page 233.
<*Makefile—Default Target*>	See "Default Rule", page 231.
<*Makefile—TWJR Targets*>	See "TWJR Targets", page 232.
<*Makefile—Utility Targets*>	See "apiutil Makefile Target", page 229.
<*Makefile—Variable Definitions*>	See "Makefile Variable Definitions", page 231.

F.1 Makefile Variable Definitions

<*Makefile—Variable Definitions*> ≡

```
# VARIABLE DEFINITIONS
######################
FILE := Ruby2_5
SHELL := /bin/bash
```

This chunk is called by {Makefile}; see its first definition at "The Makefile", page 231.

F.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 Target*> ≡

```
# DEFAULT Target
################
.PHONY : default TWJR TANGLE WEAVE TEXI INFO PDF HTML
.PHONY : twjr tangle weave texi info pdf html
default : INFO PDF HTML
```

This chunk is called by {Makefile}; see its first definition at "The Makefile", page 231.

F.3 TWJR Targets

<*Makefile—TWJR Targets*> ≡

```
# TWJR TARGETS
###############
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

INFO : info
info : $(FILE).info
$(FILE).info : $(FILE).texi
    makeinfo $(FILE).texi
openinfo : INFO
    emacs $(FILE).info

PDF : pdf
pdf : $(FILE).pdf
$(FILE).pdf : $(FILE).texi
    pdftexi2dvi --build=tidy --build-dir=build --quiet $(FILE).texi
openpdf : PDF
    open $(FILE).pdf

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

This chunk is called by {Makefile}; see its first definition at "The Makefile", page 231.

F.4 Utility Targets

<*Makefile—Utility Targets*> +≡

```
# UTILITY TARGETS
##################
```

This chunk is also defined in "apiutil Makefile Target", page 229.

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 231.

F.5 Clean Targets

<*Makefile—Clean Targets*> ≡

```
# CLEAN TARGETS
###############
.PHONY : clean veryclean dirclean distclean worldclean
clean :
    rm -vf *~ .*~ \#*\#

# clean tex detritus
texclean : clean
    rm -vf *.{dvi,aux,log,toc,cp,cps,pg,pgs}

# remove all dirs except HTML; remove *.{rb,sh} files from toplevel
dirclean : clean
    for file in *; do [ -d $$file ] && [ $${file##$(FILE)*} ] && rm -vfr $$file; done;
    rm -vf *.{rb,sh}

# remove everything except .twjr, .texi, and Makefile
distclean : dirclean
    for file in *; do [[ $$file =~ twjr|texi|Makefile ]] && : || rm -vrf $$file ; done

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

This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 231.

Appendix G Code Chunk Summaries

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

G.1 Source File Definitions

{`Makefile`}
This chunk is defined in "The Makefile", page 231.

{`apiutil.awk`}
This chunk is defined in "API Utility", page 219.

{`eval.rb`}
This chunk is defined in "Ruby Eval Utility", page 215.

{`fact.rb`}
This chunk is defined in "On Simple Examples", page 57.

{`guess.rb`}
This chunk is defined in "On Puzzle Program", page 60.

{`hola.gemspec`}
This chunk is defined in "Your First Gem", page 184.

{`hola.rb`}
This chunk is defined in "Your First Gem", page 184.

{`regx.rb`}
This chunk is defined in "Regular Expressions", page 61.

{`ri20min.rb`}
This chunk is defined in "Large Class Definition", page 37.

{`setpath.sh`}
This chunk is defined in "Initial Setup", page 230.

G.2 Code Chunk Definitions

<*Makefile—Clean Targets*>
This chunk is defined in "Clean Targets", page 233.

<*Makefile—Default Target*>
This chunk is defined in "Default Rule", page 231.

<*Makefile—TWJR Targets*>
This chunk is defined in "TWJR Targets", page 232.

<*Makefile—Utility Targets*>
Multiple definitions occur in "apiutil Makefile Target", page 229, and "Utility Targets", page 233.

<*Makefile—Variable Definitions*>
This chunk is defined in "Makefile Variable Definitions", page 231.

<*eval—EvalWrapper–Indentation Deltas*>
This chunk is defined in "`eval.rb` Indentation Deltas Code", page 216.

<*eval—Main–Get Line*>
This chunk is defined in "`eval.rb` Main Get Line Code", page 217.

<*eval—Main–Process Line*>
This chunk is defined in "`eval.rb` Main Process Line Code", page 217.

<*eval—Main–Process Line-If Not Line*>
This chunk is defined in "`eval.rb` If Not Line Code", page 218.

<*eval—Main–Process Line-Is Line*>
This chunk is defined in "`eval.rb` If Is Line Code", page 218.

<*eval—Main–Process Line-Is Line_Indentation*>
This chunk is defined in "`eval.rb` If Is Line Code", page 218.

<*eval—Main–Process Line-Is Line_Worth Evaluating?*>
This chunk is defined in "`eval.rb` If Is Line Code", page 219.

G.3 Code Chunk References

<*Makefile—Clean Targets*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 231.

<*Makefile—Default Target*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 231.

<*Makefile—TWJR Targets*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 231.

<*Makefile—Utility Targets*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 231.

<*Makefile—Variable Definitions*>
This chunk is called by {`Makefile`}; see its first definition at "The Makefile", page 231.

<*MegaGreeter—Initialize Method*>
This chunk is called by {`ri20min.rb`}; see its first definition at "Large Class Definition", page 37.

<*MegaGreeter—Main Script*>
This chunk is called by {`ri20min.rb`}; see its first definition at "Large Class Definition", page 37.

<*MegaGreeter—say_bye Method*>
This chunk is called by {`ri20min.rb`}; see its first definition at "Large Class Definition", page 37.

<*MegaGreeter—say_hi Method*>
This chunk is called by {`ri20min.rb`}; see its first definition at "Large Class Definition", page 37.

<*apiutil—BEGIN Block*>
This chunk is called by {`apiutil.awk`}; see its first definition at "API Utility", page 219.

<*apiutil—BEGIN–Ord Function Init*>
This chunk is called by <*apiutil—BEGIN Block*>; see its first definition at "`apiutil.awk` BEGIN Block", page 220.

<*apiutil—BEGIN–Variable Defns*>
This chunk is called by <*apiutil—BEGIN Block*>; see its first definition at "`apiutil.awk` BEGIN Block", page 220.

<*apiutil—BEGINFILE Block*>
This chunk is called by {`apiutil.awk`}; see its first definition at "API Utility", page 219.

<*apiutil—END Block*>
This chunk is called by {`apiutil.awk`}; see its first definition at "API Utility", page 219.

<*apiutil—ENDFILE Block*>
This chunk is called by {`apiutil.awk`}; see its first definition at "API Utility", page 219.

<*apiutil—MAIN Block*>
This chunk is called by {`apiutil.awk`}; see its first definition at "API Utility", page 219.

<*apiutil—MAIN Block–Convert Symbols Function Defn*>
This chunk is called by <*apiutil—MAIN Block*>; see its first definition at "`apiutil.awk` MAIN Block", page 220.

<*apiutil—MAIN Block–Main Loop*>
This chunk is called by <*apiutil—MAIN Block*>; see its first definition at "`apiutil.awk` MAIN Block", page 220.

<*apiutil—MAIN Block–Main Loop-Inside*>
This chunk is called by <*apiutil—MAIN Block–Main Loop*>; see its first definition at "`apiutil.awk` MAIN Block", page 221.

<*apiutil—MAIN Block–Main Loop-Inside_Processing*>
This chunk is called by <*apiutil—MAIN Block–Main Loop-Inside*>; see its first definition at "`apiutil.awk` MAIN Block", page 222.

<*apiutil—MAIN Block–Ord Function Defn*>
This chunk is called by <*apiutil—MAIN Block*>; see its first definition at "`apiutil.awk` MAIN Block", page 220.

<*eval—EvalWrapper–Constants*>
This chunk is called by {`eval.rb`}; see its first definition at "Ruby Eval Utility", page 215.

<*eval—EvalWrapper–Indentation Deltas*>
This chunk is called by {`eval.rb`}; see its first definition at "Ruby Eval Utility", page 215.

<*eval—Main–Get Line*>
This chunk is called by {`eval.rb`}; see its first definition at "Ruby Eval Utility", page 215.

<*eval—Main–Process Line*>
This chunk is called by {`eval.rb`}; see its first definition at "Ruby Eval Utility", page 215.

<*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 217.

<*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 217.

<*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 218.

<*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 218.

Bibliography

ProgrammingRuby —
Marek Hulan, Marek Jalen, Ivan Necas, *Programming Ruby*, *Version 0.2*, September 25, 2017.

List of Tables

Index

=

@

_

‘

\

{

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Program Index