Chapter 19

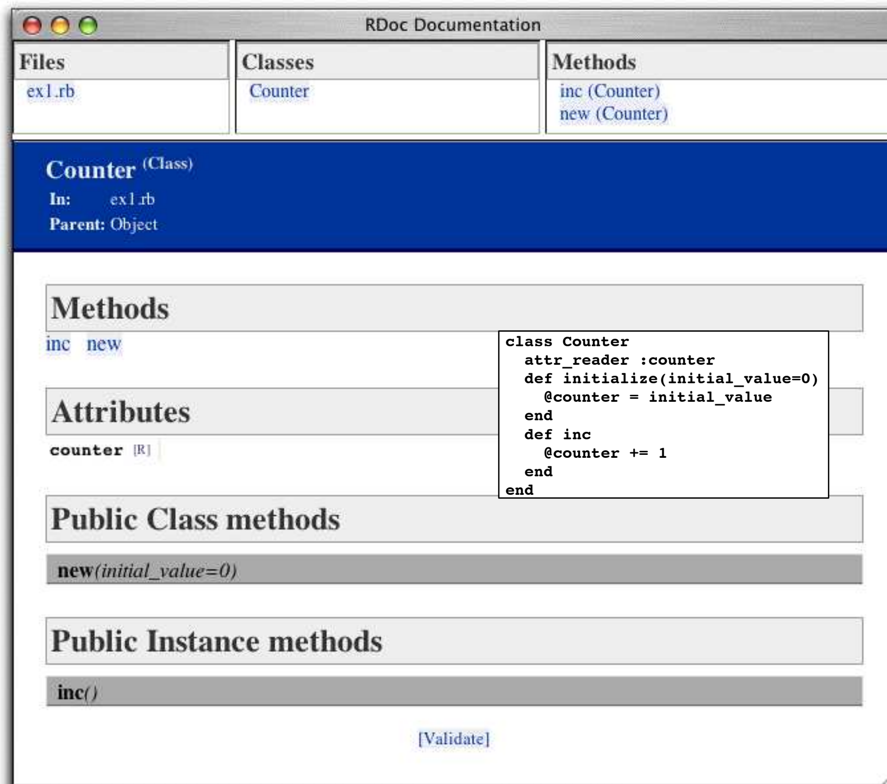
Documenting Ruby

As of version 1.8, Ruby comes bundled with RDoc, a tool that extracts and formats documentation that's embedded in Ruby source code files. This tool is used to document the built-in Ruby classes and modules. An increasing number of libraries and extensions are also documented this way.

RDoc does two jobs. First, it analyzes Ruby and C source files, looking for information to document.[1] Second, it takes this information and converts it into something readable. Out of the box, RDoc produces two kinds of output: HTML and ri. Some HTML-formatted RDoc output in a browser window is shown in Figure 19.1 on the following page. This is the result of feeding RDoc a Ruby source file with no additional documentation—RDoc does a credible job of producing something meaningful. If our source code contains comments, RDoc can use them to spice up the documentation it produces. Typically, the comment before an element is used to document that element, as shown in Figure 19.2 on page 292.

RDoc can also be used to produce documentation that can be read by the `ri` command-line utility. For example, if we ask RDoc to document the code in Figure 19.2 this way, we can then access the documentation using ri, as shown in Figure 19.3 on page 293. New Ruby distributions have the built-in classes and modules (and some libraries) documented this way. The output produced if you type `ri Proc` is shown in Figure 19.4 on page 294.

1. RDoc can also document Fortran 77 programs.



This figure shows some RDoc output in a browser window. The overlaid box shows the source program from which this output was generated. Even though the source contains no internal documentation, RDoc still manages to extract interesting information from it. We have three panes at the top of the screen showing the files, classes, and methods for which we have documentation. For class Counter, RDoc shows us the attributes and methods (including the method signatures). And if we clicked a method signature, RDoc would pop up a window containing the source code for the corresponding method.

Figure 19.1. Browse RDoc Output for Class Counter

RDoc Documentation

Files	Classes	Methods
ex2.rb	Counter	inc (Counter)
		new (Counter)

Counter (Class)
In: ex2.rb
Parent: Object

Implements a simple accumulator, whose value is accessed via the attribute *counter*. Calling the method Counter#inc increments this value.

```
# Implements a simple accumulator, whose
# value is accessed via the attribute
# _counter_. Calling the method Counter#inc
# increments this value.

class Counter

  # The current value of the count
  attr_reader :counter

  # create a new Counter with the given
  # initial value
  def initialize(initial_value=0)
    @counter = initial_value
  end

  # increment the current value of the count
  def inc
    @counter += 1
  end
end
```

Methods

inc new

Attributes

`counter` [R] The current value of the count

Public Class methods

new(*initial_value=0*)

create a new Counter with the given initial value

Public Instance methods

inc()

increment the current value of the count

[Validate]

Notice how the comments before each element now appear in the RDoc output, reformatted into HTML. Less obvious is that RDoc has detected hyperlink opportunities in our comments: in the class-level comment, the reference to `Counter#inc` is a hyperlink to the method description, and in the comment for the new method, the reference to class Counter hyperlinks back to the class documentation. This is a key feature of RDoc: it is designed to be unintrusive in the Ruby source files and to make up for this by trying to be clever when producing output.

Figure 19.2. Browse RDoc Output When Source Has Comments

Figure 19.3. Using ri to Read Documentation

```
% ri Counter
---------------------------------------- Class: Counter
     Implements a simple accumulator, whose value is
     accessed via the attribute counter. Calling the
     method Counter#inc increments this value.
-------------------------------------------------------

Class methods:
     new

Instance methods:
     inc

Attributes:
     counter

% ri Counter.inc
------------------------------------------- Counter#inc
     inc()
-------------------------------------------------------
     increment the current value of the count
```

Adding RDoc to Ruby Code

RDoc parses Ruby source files to extract the major elements (classes, modules, methods, attributes, and so on). You can choose to associate additional documentation with these by simply adding a comment block before the element in the file.

Comment blocks can be written fairly naturally, either by using # on successive lines of the comment or by including the comment in a =begin... =end block. If you use the latter form, the =begin line must be flagged with an rdoc tag to distinguish the block from other styles of documentation.

```
=begin rdoc
Calculate the minimal-cost path though the graph
using Debrinkski's algorithm, with optimized
inverse pruning of isolated leaf nodes.
=end
def calculate_path
  . . .
end
```

Within a documentation comment, paragraphs are lines that share the left margin. Text indented past this margin is formatted verbatim.

Figure 19.4. Documentation for Class `Proc` Generated by RDoc/ri

```
% ri Proc
------------------------------------------- Class: Proc
     Proc objects are blocks of code that have been
     bound to a set of local variables. Once bound,
     the code may be called in different contexts and
     still access those variables.

        def gen_times(factor)
          return Proc.new {|n| n*factor }
        end

        times3 = gen_times(3)
        times5 = gen_times(5)

        times3.call(12)               #=> 36
        times5.call(5)                #=> 25
        times3.call(times5.call(4))   #=> 60

-------------------------------------------------------

Class methods:
     new

Instance methods:
     ==, [], arity, binding, call, clone, eql?, hash,
     to_proc, to_s
```

Nonverbatim text can be marked up. To set individual words in italic, bold, or typewriter fonts, you can use _word_, *word*, and +word+, respectively. If you want to do this to multiple words or text containing nonword characters, you can use multiple words, more words, and <tt>yet more words</tt>. Putting a backslash before inline markup stops it being interpreted.

RDoc stops processing comments if it finds a comment line starting #--. This can be used to separate external from internal comments or to stop a comment being associated with a method, class, attribute, or module. Documenting can be turned back on by starting a line with #++:

```
# Extract the age and calculate the
# date of birth.
#--
# FIXME: fails if the birthday falls on
# February 29th, or if the person
# was born before epoch and the installed
# Ruby doesn't support negative time_t
#++
```

```
# The DOB is returned as a Time object.
#--
# But should probably change to use Date.
def get_dob(person)
           ...
end
```

Hyperlinks

Names of classes, source files, and any method names containing an underscore or preceded by a hash character are automatically hyperlinked from comment text to their description.

Hyperlinks to the 'net starting http:, mailto:, ftp:, and www: are recognized. An HTTP URL that references an external image file is converted into an inline <IMG...> tag. Hyperlinks starting link: are assumed to refer to local files whose paths are relative to the --op directory, where output files are stored.

Hyperlinks can also be of the form `label[url]`, in which case the label is used in the displayed text and `url` is used as the target. If the label contains multiple words, surround it in braces: `{two words}[url]`.

Lists

Lists are typed as indented paragraphs with

- A * or - (for bullet lists)
- A digit followed by a period for numbered lists
- An uppercase or lowercase letter followed by a period for alpha lists

For example, you could produce something like the previous text with this:

```
#  Lists are typed as indented paragraphs with
#  * a * or - (for bullet lists),
#  * a digit followed by a period for
#    numbered lists,
#  * an uppercase or lowercase letter followed
#    by a period for alpha lists.
```

Note how subsequent lines in a list item are indented to line up with the text in the element's first line.

Labeled lists (sometimes called *description lists*) are typed using square brackets for the label:

```
#  [cat]    Small domestic animal
#  [+cat+]  Command to copy standard input
#           to standard output
```

Labeled lists may also be produced by putting a double colon after the label. This sets the result in tabular form so the descriptions all line up in the output.

```
#  cat::   Small domestic animal
#  +cat+:: Command to copy standard input
#          to standard output
```

For both kinds of labeled lists, if the body text starts on the same line as the label, then the start of that text determines the block indent for the rest of the body. The text may also start on the line following the label, indented from the start of the label. This is often preferable if the label is long. Both of the following are valid labeled list entries:

```
#  <tt>--output</tt> <i>name [, name]</i>::
#      specify the name of one or more output files. If multiple
#      files are present, the first is used as the index.
#
#  <tt>--quiet:</tt>:: do not output the names, sizes, byte counts,
#                      index areas, or bit ratios of units as
#                      they are processed.
```

Headings

Headings are entered on lines starting with equals signs. The more equals signs, the higher the level of heading:

```
#  = Level One Heading
#  == Level Two Heading
#  and so on...
```

Rules (horizontal lines) are entered using three or more hyphens:

```
#  and so it goes...
#  ----
#  The next section...
```

Documentation Modifiers

Method parameter lists are extracted and displayed with the method description. If a method calls `yield`, then the parameters passed to `yield` will also be displayed. For example, consider the following code:

```
def fred
  ...
  yield line, address
```

This will get documented as follows:

```
fred() {|line, address| ... }
```

You can override this using a comment containing :yields: ... on the same line as the method definition:

```
def fred      # :yields: index, position
  ...
  yield line, address
```

which will get documented as follows:

```
fred() {|index, position| ... }
```

:yields: is an example of a documentation modifier. These appear immediately after the start of the document element they are modifying.

Other modifiers include the following:

`:nodoc:` *[all]*
: Don't include this element in the documentation. For classes and modules, the methods, aliases, constants, and attributes directly within the affected class or module will also be omitted from the documentation. By default, though, modules and classes within that class or module will be documented. This is turned off by adding the all modifier. For example, in the following code, only class SM::Input will be documented:

```
module SM  #:nodoc:
  class Input
  end
end
module Markup #:nodoc: all
  class Output
  end
end
```

`:doc:`
: This forces a method or attribute to be documented even if it wouldn't otherwise be. This is useful if, for example, you want to include documentation of a particular private method.

`:notnew:`
: (Applicable only to the initialize instance method.) Normally RDoc assumes that the documentation and parameters for #initialize are actually for the corresponding class's new method and so fakes out a new method for the class. The `:notnew:` modifier stops this. Remember that #initialize is protected, so you won't see the documentation unless you use the -a command-line option.

Other Directives

Comment blocks can contain other directives:

`:call-seq:` *lines...*
: Text up to the next blank comment line is used as the calling sequence when generating documentation (overriding the parsing of the method parameter list). A line is considered blank even if it starts with #. For this one directive, the leading colon is optional.

`:include:` *filename*
: This includes the contents of the named file at this point. The file will be searched for in the directories listed by the --include option or in the current directory by default. The contents of the file will be shifted to have the same indentation as the : at the start of the :include: directive.

`:title:` *text*
: This sets the title for the document. It's equivalent to the --title command-line parameter. (The command-line parameter overrides any :title: directive in the source.)

`:main:` *name*
This is equivalent to the --main command-line parameter, setting the initial page displayed for this documentation.

`:stopdoc:` / `:startdoc:`
This stops and starts adding new documentation elements to the current container. For example, if a class has a number of constants that you don't want to document, put a `:stopdoc:` before the first and a `:startdoc:` after the last. If you don't specify a `:startdoc:` by the end of the container, this disables documentation for the entire class or module.

`:enddoc:`
This documents nothing further at the current lexical level.

A larger example of a file documented using RDoc is shown in Figure 19.5 on page 302.

Adding RDoc to C Extensions

RDoc understands many of the conventions used when writing extensions to Ruby in C.

If RDoc sees a C function named `Init_`*Classname*, it treats it as a class definition—any C comment before the `Init_` function will be used as the class's documentation.

The `Init_` function is normally used to associate C functions with Ruby method names. For example, a Cipher extension may define a Ruby method salt=, implemented by the C function salt_set using a call such as this:

```
rb_define_method(cCipher, "salt=", salt_set, 1);
```

RDoc parses this call, adding the salt= method to the class documentation. RDoc then searches the C source for the C function salt_set. If this function is preceded by a comment block, RDoc uses this for the method's documentation.

This basic scheme works with no effort on your part beyond writing the normal documentation in the comments for functions. However, RDoc cannot discern the calling sequence for the corresponding Ruby method. In this example, the RDoc output will show a single argument with the (somewhat meaningless) name "arg1." You can override this using the call-seq directive in the function's comment. The lines following call-seq (up to a blank line) are used to document the calling sequence of the method:

```
/*
 * call-seq:
 *   cipher.salt = number
 *   cipher.salt = "string"
 *
 * Sets the salt of this cipher to either a binary +number+ or
 * bits in +string+.
 */
static VALUE
salt_set(cipher, salt)
...
```

If a method returns a meaningful value, it should be documented in the call-seq following the characters ->:

```
/*
 * call-seq:
 *   cipher.keylen   -> Fixnum or nil
 */
```

Although RDoc heuristics work well for finding the class and method comments for simple extensions, they don't always work for more complex implementations. In these cases, you can use the directives Document-class: and Document-method: to indicate that a C comment relates to a given class or method, respectively. The modifiers take the name of the Ruby class or method that's being documented:

```
/*
 * Document-method: reset
 *
 * Clear the current buffer and prepare to add new
 * cipher text. Any accumulated output cipher text
 * is also cleared.
 */
```

Finally, it is possible in the Init_xxx function to associate a Ruby method with a C function in a different C source file. RDoc would not find this function without your help: you add a reference to the file containing the function definition by adding a special comment to the rb_define_method call. The following example tells RDoc to look in the file `md5.c` for the function (and related comment) corresponding to the md5 method:

```
rb_define_method(cCipher, "md5", gen_md5, -1); /* in md5.c */
```

A C source file documented using RDoc is shown in Figure 19.6 on page 303. Note that the bodies of several internal methods have been elided to save space.

Running RDoc

You run RDoc from the command line:

```
% rdoc [options] [filenames...]
```

Type **rdoc --help** for an up-to-date option summary.

Files are parsed, and the information they contain collected, before any output is produced. This allows cross-references between all files to be resolved. If a name is a directory, it is traversed. If no names are specified, all Ruby files in the current directory (and subdirectories) are processed.

A typical use may be to generate documentation for a package of Ruby source (such as RDoc itself):

```
% rdoc
```

This command generates HTML 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/`.

RDoc uses file extensions to determine how to process each file. Filenames ending `.rb` and `.rbw` are assumed to be Ruby source. Filenames ending `.c` are parsed as C files. All other files are assumed to contain just markup (with or without leading `#` comment markers). If directory names are passed to RDoc, they are scanned recursively for source files only. To include nonsource files such as `README`s in the documentation process, their names must be given explicitly on the command line.

When writing a Ruby library, you often have some source files that implement the public interface, but the majority are internal and of no interest to the readers of your documentation. In these cases, construct a `.document` file in each of your project's directories. If RDoc enters a directory containing a `.document` file, it will process only the files in that directory whose names match one of the lines in that file. Each line in the file can be a filename, a directory name, or a wildcard (a file system "glob" pattern). For example, to include all Ruby files whose names start main, along with the file `constants.rb`, you could use a `.document` file containing this:

```
main*.rb
constants.rb
```

Some project standards ask for documentation in a top-level README file. You may find it convenient to write this file in RDoc format and then use the :include: directive to incorporate the README into the documentation for the main class.

Create Documentation for ri

RDoc is also used to create documentation, which will be later displayed using ri.

When you run ri, it by default looks for documentation in three places:[2]

- The *system* documentation directory, which holds the documentation distributed with Ruby and which is created by the Ruby install process
- The *site* directory, which contains sitewide documentation added locally
- The *user* documentation directory, stored under the user's own home directory

You can find these three directories in the following locations:

- `$datadir/ri/<ver>/system/...`
- `$datadir/ri/<ver>/site/...`
- `~/.rdoc/....`

The variable `$datadir` is the configured data directory for the installed Ruby. Find your local *datadir* using this:

```
ruby -r rbconfig -e 'p Config::CONFIG["datadir"]'
```

2. You can override the directory location using the `--op` option to RDoc and subsequently using the `--doc-dir` option with ri.

To add documentation to ri, you need to tell RDoc which output directory to use. For your own use, it's easiest to use the --ri option, which installs the documentation into ~/.rdoc:

```
% rdoc --ri  file1.rb file2.rb
```

If you want to install sitewide documentation, use the --ri-site option:

```
% rdoc --ri-site  file1.rb file2.rb
```

The --ri-system option is normally used only to install documentation for Ruby's built-in classes and standard libraries. You can regenerate this documentation from the Ruby source distribution (not from the installed libraries themselves):

```
% cd <ruby source base>/lib
% rdoc --ri-system
```

Figure 19.5. Ruby Source File Documented with RDoc

```
# This module encapsulates functionality related to the
# generation of Fibonacci sequences.
#--
# Copyright (c) 2004 Dave Thomas, The Pragmatic Programmers, LLC.
# Licensed under the same terms as Ruby. No warranty is provided.
module Fibonacci

  # Calculate the first _count_ Fibonacci numbers, starting with 1,1.
  #
  # :call-seq:
  #   Fibonacci.sequence(count)                -> array
  #   Fibonacci.sequence(count) {|val| ... }   -> nil
  #
  # If a block is given, supply successive values to the block and
  # return +nil+, otherwise return all values as an array.
  def Fibonacci.sequence(count, &block)
    result, block = setup_optional_block(block)
    generate do |val|
      break if count <= 0
      count -= 1
      block[val]
    end
    result
  end

  # Calculate the Fibonacci numbers up to and including _max_.
  #
  # :call-seq:
  #   Fibonacci.upto(max)                -> array
  #   Fibonacci.upto(max) {|val| ... }   -> nil
  #
  # If a block is given, supply successive values to the
  # block and return +nil+, otherwise return all values as an array.
  def Fibonacci.upto(max, &block)
    result, block = setup_optional_block(block)
    generate do |val|
      break if val > max
      block[val]
    end
    result
  end

  private

  # Yield a sequence of Fibonacci numbers to a block.
  def Fibonacci.generate
    f1, f2 = 1, 1
    loop do
      yield f1
      f1, f2 = f2, f1+f2
    end
  end

  # If a block parameter is given, use it, otherwise accumulate into an
  # array. Return the result value and the block to use.
  def Fibonacci.setup_optional_block(block)
    if block.nil?
      [ result = [], lambda {|val| result << val } ]
    else
      [ nil, block ]
    end
  end
end
```

Figure 19.6. C Source File Documented with RDoc

```
#include "ruby.h"
#include "cdjukebox.h"

static VALUE cCDPlayer;
static void cd_free(void *p) { ... }
static VALUE cd_alloc(VALUE klass) { ... }
static void progress(CDJukebox *rec, int percent) { ... }

/* call-seq:
 *    CDPlayer.new(unit)  -> new_cd_player
 *
 * Assign the newly created CDPlayer to a particular unit
 */
static VALUE cd_initialize(VALUE self, VALUE unit) {
  int unit_id;
  CDJukebox *jb;

  Data_Get_Struct(self, CDJukebox, jb);

  unit_id = NUM2INT(unit);
  assign_jukebox(jb, unit_id);

  return self;
}

/* call-seq:
 *   player.seek(int_disc, int_track)  -> nil
 *   player.seek(int_disc, int_track) {|percent| } -> nil
 *
 * Seek to a given part of the track, invoking the block
 * with the percent complete as we go.
 */
static VALUE
cd_seek(VALUE self, VALUE disc, VALUE track) {
  CDJukebox *jb;
  Data_Get_Struct(self, CDJukebox, jb);

  jukebox_seek(jb, NUM2INT(disc), NUM2INT(track), progress);
  return Qnil;
}

/* call-seq:
 *   player.seek_time -> Float
 *
 * Return the average seek time for this unit (in seconds)
 */
static VALUE
cd_seek_time(VALUE self)
{
  double tm;
  CDJukebox *jb;
  Data_Get_Struct(self, CDJukebox, jb);
  tm = get_avg_seek_time(jb);
  return rb_float_new(tm);
}

/* Interface to the Spinzalot[http://spinzalot.cd]
 * CD Player library.
 */

void Init_CDPlayer() {
  cCDPlayer = rb_define_class("CDPlayer", rb_cObject);
  rb_define_alloc_func(cCDPlayer, cd_alloc);
  rb_define_method(cCDPlayer, "initialize", cd_initialize, 1);
  rb_define_method(cCDPlayer, "seek", cd_seek, 2);
  rb_define_method(cCDPlayer, "seek_time", cd_seek_time, 0);
}
```