Chapter 15

Ruby and Its World

It's an unfortunate fact of life that our applications have to deal with the big, bad world. In this chapter, we'll look at how Ruby interacts with its environment. Microsoft Windows users will probably also want to look at platform-specific information beginning on page 316.

Command-Line Arguments

"In the beginning was the command line."[1] Regardless of the system in which Ruby is deployed, whether it be a super high-end scientific graphics workstation or an embedded PDA device, you have to start the Ruby interpreter somehow, and that gives us the opportunity to pass in command-line arguments.

A Ruby command line consists of three parts: options to the Ruby interpreter, optionally the name of a program to run, and optionally a set of arguments for that program:

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

The Ruby options are terminated by the first word on the command line that doesn't start with a hyphen or by the special flag -- (two hyphens).

If no filename is present on the command line or if the filename is a single hyphen (-), Ruby reads the program source from standard input.

Arguments for the program itself follow the program name. For example, the following:

```
% ruby -w - "Hello World"
```

will enable warnings, read a program from standard input, and pass it the string "Hello World" as an argument.

1. This is the title of a marvelous essay by Neal Stephenson (available online at http://www.spack.org/index.cgi/InTheBeginningWasTheCommandLine).

Command-Line Options

-0[*octal*]
: The 0 flag (the digit zero) specifies the record separator character (\0, if no digit follows). -00 indicates paragraph mode: records are separated by two successive default record separator characters. -0777 reads the entire file at once (as it is an illegal character). Sets $/.

-a
: Autosplit mode when used with -n or -p; equivalent to executing $F = $_.split at the top of each loop iteration.

-C *directory*
: Changes working directory to *directory* before executing.

-c
: Checks syntax only; does not execute the program.

--copyright
: Prints the copyright notice and exits.

-d, --debug
: Sets $DEBUG and $VERBOSE to true. This can be used by your programs to enable additional tracing.

--disable-gems
: 1.9 Stops Ruby automatically loading RubyGems from require.

-E *encoding*, --encoding *encoding*, --encoding=*encoding*
: 1.9 Specifies the default character encoding for data read from and written to the outside world. This can be used to set both the external encoding (the encoding to be assumed for file contents) and optionally the default internal encoding (the file contents are transcoded to this when read and transcoded from this when written). The format of the *encoding* parameter is -E external, -E external:internal, or -E :internal. See 17 on page 264 for details. See also -U.

-e '*command*'
: Executes *command* as one line of Ruby source. Several -e's are allowed, and the commands are treated as multiple lines in the same program. If *programfile* is omitted when -e is present, execution stops after the -e commands have been run. Programs run using -e have access to the old behavior of ranges and regular expressions in conditions—ranges of integers compare against the current input line number, and regular expressions match against $_.

-F *pattern*
: Specifies the input field separator ($;) used as the default for split() (affects the -a option).

-h, --help
: Displays a short help screen.

-I *directories*
: Specifies directories to be prepended to $LOAD_PATH ($:). Multiple -I options may be present. Multiple directories may appear following each -I, separated by a colon (:) on Unix-like systems and by a semicolon (;) on DOS/Windows systems.

-i [*extension*]
: Edits ARGV files in place. For each file named in ARGV, anything you write to standard output will be saved back as the contents of that file. A backup copy of the file will be made if *extension* is supplied.

```
% ruby -pi.bak -e "gsub(/Perl/, 'Ruby')" *.txt
```

-l Enables automatic line-ending processing; sets $\ to the value of $/ and chops every input line automatically.

-n Assumes a while gets; . . . ; end loop around your program. For example, a simple grep command could be implemented as follows:

```
% ruby -n -e "print if /wombat/"  *.txt
```

-p Places your program code within the loop while gets; . . . ; print; end.

```
% ruby -p -e "$_.downcase!" *.txt
```

-r *library*
1.9 requires the named library or gem before executing.

-S Looks for the program file using the RUBYPATH or PATH environment variable.

-s Any command-line switches found after the program filename, but before any filename arguments or before a --, are removed from ARGV and set to a global variable named for the switch. In the following example, the effect of this would be to set the variable $opt to "electric":

```
% ruby -s prog -opt=electric ./mydata
```

-T[*level*]
1.9 Sets the safe level, which among other things enables tainting and untrusted checks (see page 436). Sets $SAFE.

-U Sets the default internal encoding to UTF-8. See 17 on page 264 for details. See also -E.

-v, --verbose
: Sets $VERBOSE to true, which enables verbose mode. Also prints the version number. In verbose mode, compilation warnings are printed. If no program filename appears on the command line, Ruby exits.

--version
: Displays the Ruby version number and exits.

-w Enables verbose mode. Unlike -v, reads program from standard input if no program files are present on the command line. We recommend running your Ruby programs with -w.

-W *level*
: Sets the level of warnings issued. With a *level* or two (or with no level specified), equivalent to -w—additional warnings are given. If *level* is 1, runs at the standard (default) warning level. With -W0, absolutely no warnings are given (including those issued using Kernel.warn).

-X *directory*
: Changes working directory to *directory* before executing. This is the same as -C *directory*.

-x [*directory*]
: Strips off text before #!ruby line and changes working directory to *directory* if given.

-y, --yydebug
: Enables yacc debugging in the parser *(waaay too much information)*.

ARGV

Any command-line arguments after the program filename are available to your Ruby program in the global array ARGV. For instance, assume `test.rb` contains the following program:

```
ARGV.each {|arg| p arg }
```

Invoke it with the following command line:

```
% ruby -w test.rb "Hello World" a1 1.6180
```

It'll generate the following output:

```
"Hello World"
"a1"
"1.6180"
```

There's a gotcha here for all you C programmers—ARGV[0] is the first argument to the program, not the program name. The name of the current program is available in the global variable $0, which is aliased to $PROGRAM_NAME. Notice that all the values in ARGV are strings. 1.9

If your program reads from standard input (or uses the special object ARGF, described on page 342), the program arguments in ARGV will be taken to be filenames, and Ruby will read from these files. If your program takes a mixture of arguments and filenames, make sure you empty the nonfilename arguments from the ARGV array before reading from the files.

Program Termination

The method Kernel#exit terminates your program, returning a status value to the operating system. However, unlike some languages, exit doesn't terminate the program immediately. Kernel#exit first raises a SystemExit exception, which you may catch, and then performs a number of cleanup actions, including running any registered at_exit methods and object finalizers. See the reference for Kernel#exit beginning on page 569 for details.

Environment Variables

You can access operating system environment variables using the predefined variable ENV. It responds to the same methods as Hash.[2]

```
ENV['SHELL']     # =>   "/bin/bash"
ENV['HOME']      # =>   "/Users/dave"
ENV['USER']      # =>   "dave"
ENV.keys.size    # =>   38
ENV.keys[0, 4]   # =>   ["MANPATH", "TERM_PROGRAM", "SHELL", "TERM"]
```

The values of some environment variables are read by Ruby when it first starts. These variables modify the behavior of the interpreter, as shown in Table 15.1 on the following page.

Writing to Environment Variables

A Ruby program may write to the ENV object. On most systems, this changes the values of the corresponding environment variables. However, this change is local to the process that makes it and to any subsequently spawned child processes. This inheritance of environment variables is illustrated in the code that follows. A subprocess changes an environment variable, and this change is inherited by a process that it then starts. However, the change is not visible to the original parent. (This just goes to prove that parents never really know what their children are doing.)

1.9 As of Ruby 1.9, setting an environment variable's value to nil removes the variable from the environment:

Download **samples/rubyworld_3.rb**

```
puts "In parent, term = #{ENV['TERM']}"
fork do
  puts "Start of child 1, term = #{ENV['TERM']}"
  ENV['TERM'] = "ansi"
  fork do
    puts "Start of child 2, term = #{ENV['TERM']}"
  end
  Process.wait
  puts "End of child 1, term = #{ENV['TERM']}"
end
Process.wait
puts "Back in parent, term = #{ENV['TERM']}"
```

produces:

```
In parent, term = xterm-color
Start of child 1, term = xterm-color
Start of child 2, term = ansi
End of child 1, term = ansi
Back in parent, term = xterm-color
```

2. ENV is not actually a hash, but if you need to, you can convert it into a Hash using ENV#to_hash.

Table 15.1. Environment Variables Used by Ruby

Variable Name	**Description**
DLN_LIBRARY_PATH	Specifies the search path for dynamically loaded modules.
HOME	Points to user's home directory. This is used when expanding ~ in file and directory names.
LOGDIR	Specifies the fallback pointer to the user's home directory if $HOME is not set. This is used only by Dir.chdir.
OPENSSL_CONF	Specifies the location of OpenSSL configuration file.
RUBYLIB	Specifies an additional search path for Ruby programs ($SAFE must be 0).
RUBYLIB_PREFIX	(Windows only) Mangles the RUBYLIB search path by adding this prefix to each component.
RUBYOPT	Specifies additional command-line options to Ruby; examined after real command-line options are parsed ($SAFE must be 0).
RUBYPATH	With -S option, specifies the search path for Ruby programs (defaults to PATH).
RUBYSHELL	Specifies shell to use when spawning a process under Windows; if not set, will also check SHELL or COMSPEC.
RUBY_TCL_DLL	Overrides default name for TCL shared library or DLL.
RUBY_TK_DLL	Overrides default name for Tk shared library or DLL. Both this and RUBY_TCL_DLL must be set for either to be used.

Where Ruby Finds Its Libraries 1.9

You use require or load to bring a library into your Ruby program. Some of these libraries are supplied with Ruby, some you may have installed from the Ruby Application Archive, some may have been packaged as RubyGems (of which more later), and some you may have written yourself. How does Ruby find them?

Let's start with the basics. When Ruby is built for your particular machine, it predefines a set of standard directories to hold library stuff. Where these are depends on the machine in question. You can determine this from the command line with something like this:

```
% ruby -e 'puts $:'
```

On my OS X box, this produces the following list (note that I have my Ruby installed in a nonstandard place while I'm writing this book):

```
/usr/local/rubybook/lib/ruby/gems/1.9.0/gems/BlueCloth-1.0.0/lib
/usr/local/rubybook/lib/ruby/gems/1.9.0/gems/BlueCloth-1.0.0/bin
/usr/local/rubybook/lib/ruby/site_ruby/1.9
/usr/local/rubybook/lib/ruby/site_ruby/1.9.0/i686-darwin8.11.1
/usr/local/rubybook/lib/ruby/site_ruby
/usr/local/rubybook/lib/ruby/vendor_ruby/1.9
/usr/local/rubybook/lib/ruby/vendor_ruby/1.9.0/i686-darwin8.11.1
/usr/local/rubybook/lib/ruby/vendor_ruby /usr/local/rubybook/lib/ruby/1.9
/usr/local/rubybook/lib/ruby/1.9.0/i686-darwin8.11.1
```

Let's skip the gems directory for now.

The site_ruby directories are intended to hold modules and extensions that you've added. The architecture-dependent directories (i686-darwin8.11.1 in this case) hold executables and other things specific to this particular machine. All these directories are automatically included in Ruby's search for libraries.

Sometimes this isn't enough. Perhaps you're working on a large project written in Ruby and you and your colleagues have built a substantial library of Ruby code. You want everyone on the team to have access to all this code. You have a couple of options to accomplish this. If your program runs at a safe level of zero (see Chapter 26 beginning on page 436), you can set the environment variable RUBYLIB to a list of one or more directories to be searched.[3] If your program is not *setuid*, you can use the command-line parameter -I to do the same thing.

The Ruby variable $: is an array of places to search for loaded files. As we've seen, this variable is initialized to the list of standard directories, plus any additional ones you specified using RUBYLIB and -I. You can always add directories to this array from within your running program.

1.9 RubyGems Integration

This section is based on the start of the chapter on RubyGems written by Chad Fowler for the second edition of this book.

RubyGems is a standardized packaging and installation framework for Ruby libraries and applications. RubyGems makes it easy to locate, install, upgrade, and uninstall Ruby packages.

Before RubyGems came along, installing a new library involved searching the Web, downloading a package, and attempting to install it—only to find that its dependencies haven't been met. If the library you want is packaged using RubyGems, however, you can now simply ask RubyGems to install it (and all its dependencies). Everything is done for you.

In the RubyGems world, developers bundle their applications and libraries into single files called *gems*. These files conform to a standardized format and typically are stored in repositories on the 'net (but you can also create your own repositories if you want).

The RubyGems system provides a command-line tool, appropriately named gem, for manipulating these gem files. It also provides integration into Ruby so that your programs can access gems as libraries.

Prior to Ruby 1.9, it was your responsibility to install the RubyGems software on your computer. Now, however, Ruby comes with RubyGems baked right in.

3. The separator between entries depends on your platform. For Windows, it's a semicolon; for Unix, it's a colon.

Installing Gems on Your Machine

Your latest project calls for a lot of XML generation. You could just hard-code it, but you've heard great things about Jim Weirich's Builder library, which lets you construct XML directly from Ruby code.

Let's start by seeing whether Builder is available as a gem:

```
% gem query --details --remote --name-matches build
*** REMOTE GEMS ***

AntBuilder (0.4.3)
    Author: JRuby-extras
    Homepage: http://jruby-extras.rubyforge.org/

    AntBuilder: Use ant from JRuby. Only usable within JRuby

builder (2.1.2)
    Author: Jim Weirich
    Homepage: http://onestepback.org

    Builders for MarkUp.

...
```

The --details option displays the description of any gems it finds. The --remote option searches the remote repository. And the --name-matches option says to search the central gem repository for any gem whose name matches the regular expression /build/. (We could have used the short-form options -d, -r, and -n.) The result shows a number of gems have *build* in their name; the one we want is just plain Builder.

The number after the name shows the latest version. You can see a list of all available versions using the --all option. We'll also use the list command, because it lets us match on an exact name:

```
% gem list --details --remote --all builder
*** REMOTE GEMS ***

builder (2.1.2, 2.1.1, 2.0.0, 1.2.4, 1.2.3, 1.2.2, 1.2.1, 1.2.0, 1.1.0,
         1.0.0, 0.1.1, 0.1.0)
    Author: Jim Weirich
    Homepage: http://onestepback.org

    Builders for MarkUp.
```

Because we want to install the most recent one, we don't have to state an explicit version on the install command; the latest is downloaded by default:

```
% gem install builder
Successfully installed builder-2.1.2
1 gem installed
Installing ri documentation for builder-2.1.2...
ERROR:  While generating documentation for builder-2.1.2
```

```
... MESSAGE:   Unhandled special: Special: type=17, text="<!-- HI -->"
... RDOC args: --ri --op /usr/local/rubybook/lib/ruby/gems/1.9.0/...
(continuing with the rest of the installation)
Installing RDoc documentation for builder-2.1.2...
...
```

Several things happened here. First, we see that the latest version of the Builder gem (2.1.2) has been installed. Next we see that RubyGems has determined that Jim has created documentation for his gem, so it sets about extracting it using RDoc. During the extraction, RDoc encounters a construct it can't handle and complains. You'll see this happen every now and then. It's annoying, but you can ignore the message.

If you're running gem install on a Unix platform, you'll need to prefix the command with sudo, because by default the local gems are installed into shared system directories.

During installation, you can add the -t option to the RubyGems install command, causing RubyGems to run the gem's test suite (if one has been created). If the tests fail, the installer will prompt you to either keep or discard the gem. This is a good way to gain a little more confidence that the gem you've just downloaded works on your system the way the author intended.

Let's see what gems we now have installed on our local box:

```
% gem list
*** LOCAL GEMS ***
builder (2.1.2)
```

Reading the Gem Documentation

Being that this is your first time using Builder, you're not exactly sure how to use it. Fortunately, RubyGems installed the documentation for Builder on your local machine. We just have to find it.

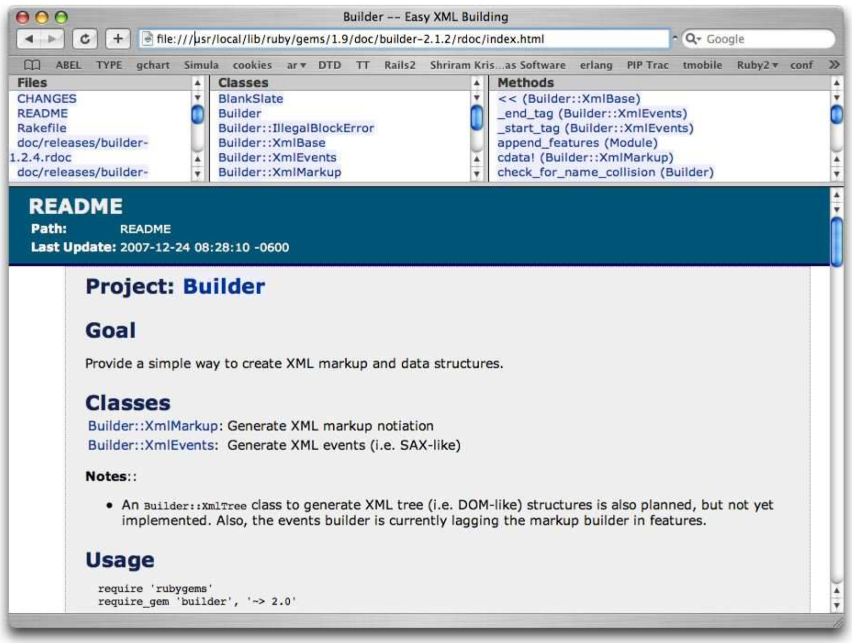
As with most things in RubyGems, the documentation for each gem is stored in a central, protected, RubyGems-specific place. This will vary by system and by where you may explicitly choose to install your gems. The most reliable way to find the documents is to ask the gem command where your RubyGems main directory is located:

```
% gem environment gemdir
/usr/local/lib/ruby/gems/1.9.0
```

RubyGems stores generated documentation beneath the `doc/` subdirectory of this directory, in this case `/usr/local/lib/ruby/gems/1.9.0/doc`. Each gem has its own documentation directory. Inside this directory, you'll find the HTML in the subdirectory `rdoc/`. You can open `index.html` and view the documentation (the full path is `/usr/local/lib/ruby/gems/1.9.0/doc/builder-2.1.2/rdoc/index.html`. The result will look something like Figure 15.1 on the following page.

If you find yourself using this path often, you can create a shortcut.

Figure 15.1. Installed Documentation for Builder


Builder -- Easy XML Building
file:///usr/local/lib/ruby/gems/1.9/doc/builder-2.1.2/rdoc/index.html
Google
ABEL TYPE gchart Simula cookies ar DTD TT Rails2 Shriram Kris...as Software erlang PIP Trac tmobile Ruby2 conf

Files
CHANGES
README
Rakefile
doc/releases/builder-1.2.4.rdoc
doc/releases/builder-

Classes
BlankSlate
Builder
Builder::IllegalBlockError
Builder::XmlBase
Builder::XmlEvents
Builder::XmlMarkup

Methods
<< (Builder::XmlBase)
_end_tag (Builder::XmlEvents)
_start_tag (Builder::XmlEvents)
append_features (Module)
cdata! (Builder::XmlMarkup)
check_for_name_collision (Builder)

README
Path: README
Last Update: 2007-12-24 08:28:10 -0600

Project: Builder

Goal

Provide a simple way to create XML markup and data structures.

Classes

Builder::XmlMarkup: Generate XML markup notiation
Builder::XmlEvents: Generate XML events (i.e. SAX-like)

Notes::

- An Builder::XmlTree class to generate XML tree (i.e. DOM-like) structures is also planned, but not yet implemented. Also, the events builder is currently lagging the markup builder in features.

Usage

```
require 'rubygems'
require_gem 'builder', '~> 2.0'
```

Here's one way to do that on Mac OS X boxes:

```
% gemdoc=`gem environment gemdir`/doc
% ls $gemdoc
builder-2.1.2
% open $gemdoc/builder-2.1.2/rdoc/index.html
```

To save time, you could declare $gemdoc in your login shell's profile or .rc file.

The second (and easier) way to view gems' RDoc documentation is to use RubyGems'
1.9 included gem server utility. To start gem server, simply type this:

```
% gem server
Starting gem server on http://localhost:8808/
```

gem server starts a web server running on whatever computer you run it on. By default, it will start on port 8808 and will serve gems and their documentation from the default RubyGems installation directory. Both the port and the gem directory are overridable via command-line options, using the -p and -d options, respectively.

Once you've started the gem server program, if you are running it on your local computer, you can access the documentation for your installed gems by pointing your web browser to http://localhost:8808. There, you will see a list of the gems you have installed with their descriptions and links to their RDoc documentation.

Using a Gem

Once a gem is installed, you use require to load it into your own code, just as you would any other Ruby library:[4]

Download **samples/rubyworld_5.rb**

```
require 'builder'
xml = Builder::XmlMarkup.new(target: STDOUT, indent: 2)
xml.person(type: "programmer") do
  xml.name do
    xml.first "Dave"
    xml.last "Thomas"
  end
  xml.location "Texas"
  xml.preference("ruby")
end
```

produces:

```
<person type="programmer">
  <name>
    <first>Dave</first>
    <last>Thomas</last>
  </name>
  <location>Texas</location>
  <preference>ruby</preference>
</person>
```

Gems and Versions

Maybe you first started using Builder a few years ago. Back then the interface was a little bit different—with versions prior to Build 1.0, you could say this:

```
xml = Builder::XmlMarkup.new(STDOUT, 2)
xml.person do
  name("Dave Thomas")
  location("Texas")
end
```

Note that the constructor takes positional parameters. Also, in the do block, we can say just name(...) (whereas the current Builder requires xml.name(...)).

4. Prior to Ruby 1.9, before you could use a gem in your code, you first had to load a support library called rubygems. Ruby now integrates that support directly, so this step is no longer needed.

We could go through our old code and update it all to work with the new-style Builder—that's probably the best long-term solution. But we can also let RubyGems handle the issue for us.

When we asked for a listing of the Builder gems in the repository, we saw that multiple versions were available:

```
% gem list -ra builder
*** REMOTE GEMS ***
builder (2.1.2, 2.1.1, 2.0.0, 1.2.4, 1.2.3, 1.2.2, 1.2.1,
1.2.0, 1.1.0, 1.0.0, 0.1.1, 0.1.0)
```

When we installed Builder previously, we didn't specify a version, so RubyGems automatically installed the latest. But we can also get it to install a specific version or a version meeting some given criteria. Let's install the most recent release of Builder whose version number is less than 1:

```
% gem install builder --version '< 1'
Successfully installed builder-0.1.1
1 gem installed
Installing ri documentation for builder-0.1.1...
Installing RDoc documentation for builder-0.1.1...
```

Have we just overwritten the 2.1.2 release of Builder that we'd previously installed? Let's find out by listing our locally installed gems:

```
% gem list builder
*** LOCAL GEMS ***
builder (2.1.2, 0.1.1)
```

Now that we have both versions installed locally, how do we tell our legacy code to use the old one while still having our new code use the latest version? It turns out that require automatically loads the latest version of a gem, so the code from page 243 will work fine. If we want to specify a version number when we load a gem, we have to do a little bit more work, making it explicit that we're using RubyGems:

```
gem 'builder', '< 1.0'
require 'builder'
xml = Builder::XmlMarkup.new(STDOUT, 2)
xml.person do
  name("Dave Thomas")
  location("Texas")
end
```

The magic is the gem line, which says, "When looking for the builder gem, consider only those versions less than 1.0." The subsequent require honors this, so the code loads the correct version of Builder and runs. The '< 1.0' part of the gem line is a version predicate. Table 15.2 on page 246 shows the various predicates that RubyGems supports. You can specify multiple version predicates, so the following is valid:

```
gem 'builder', '> 0.1', '< 0.1.5'
```

Unfortunately, after all this work, there's a problem. Older versions of Builder don't run under 1.9 anyway. You can still run this code in Ruby 1.8, but you'd have to update your code to use the new-style Builder if you want to use Ruby 1.9.

Gems Can Be More Than Libraries

As well as installing libraries that can be used inside your application code, RubyGems can also install utility programs that you can invoke from the command line. Often these utilities are wrappers around the libraries included in the gem. For example, Marcel Molina's AWS:S3 gem is a library that gives you programmatic access to Amazon's S3 storage facility. As well as the library itself, Marcel provided a command-line utility, s3sh, which lets you interact with your S3 assets. When you install the gem, s3sh is automatically loaded into the same bin/ directory that holds the Ruby interpreter.

There's a small problem with these installed utilities. Although gems supports versioning of libraries, it does not version command-line utilities. With these, it's "last one in wins."

The Rake Build Tool

As well as the Builder gem, Jim Weirich wrote an incredibly useful utility program called Rake. Prior to Ruby 1.9, you had to install Rake as a separate gem, but it is now included in the base Ruby installation. 1.9

Rake was initially implemented as a Ruby version of Make, the common build utility. However, calling Rake a build utility is to miss its true power. Really, Rake is an automation tool—it's a way of putting all those tasks that you perform in a project into one neat and tidy place.

Let's start with a trivial example. As you edit files, you often accumulate backup files in your working directories. On Unix systems, these files often have the same name as the original files, but with a tilde character appended. On Windows boxes, the files often have a .bak extension.

We could write a trivial Ruby program that deletes these files. For a Unix box, it might look something like this:

```
require 'fileutils'
files = Dir['*~']
FileUtils::rm files, verbose: true
```

The FileUtils module defines methods for manipulating files and directories (see the description on page 755). Our code uses its rm method. We use the Dir class to return a list of filenames matching the given pattern and pass that list to rm.

Let's package this code as a *task*—a chunk of code that Rake can execute for us.

By default, Rake searches the current directory (and its parents) for a file called Rakefile. This file contains definitions for the tasks that Rake can run.

Table 15.2. Version Operators

Both the gem method and the add_dependency attribute in a Gem::Specification accept arguments that specify a version dependency. RubyGems version dependencies are of the form operator major.minor.patch_level. Listed next is a table of all the possible version operators.

Operator	Description
=	Exact version match. Major, minor, and patch level must be identical.
!=	Any version that is not the one specified.
>	Any version that is greater (even at the patch level) than the one specified.
<	Any version that is less than the one specified.
>=	Any version greater than or equal to the specified version.
<=	Any version less than or equal to the specified version.
~>	“Boxed” version operator. Version must be greater than or equal to the specified version *and* less than the specified version after having its minor version number increased by 1. This is to avoid API incompatibilities between minor version releases.

So, put the following code into a file called `Rakefile`:

```
desc "Remove files whose names end with a tilde"
task :delete_unix_backups do
  files = Dir['*~']
  rm(files, verbose: true) unless files.empty?
end
```

Although it doesn’t have an `.rb` extension, this is actually just a file of Ruby code. Rake defines an environment containing methods such as desc and task and then executes the `Rakefile`.

The desc method provides a single line of documentation for the task that follows it. The task method defines a Rake task that can be executed from the command line. The parameter is the name of the task (a symbol), and the block that follows is the code to be executed. Here we can just use rm—all the methods in FileUtils are automatically available inside Rake files.

We can invoke this task from the command line:

```
% rake delete_unix_backups
(in /Users/dave/BS2/titles/RUBY3/Book/code/rake)
rm entry~
```

The first line shows us the name of the directory where Rake found the `Rakefile` (remember that this might be in a directory above our current working directory). The next line is the output of the rm method, in this case showing it deleted the single file `entry~`.

OK, now let’s write a second task in the same `Rakefile`. This one deletes Windows backup files.

```
desc "Remove files whose names end with a tilde"
task :delete_unix_backups do
  files = Dir['*~']
  rm(files, verbose: true) unless files.empty?
end
desc "Remove files with a .bak extension"
task :delete_windows_backups do
  files = Dir['*.bak']
  rm(files, verbose: true) unless files.empty?
end
```

We can run this with rake delete_windows_backups.

But let's say that our application could be used on both platforms, and we wanted to let our users delete backup files on either. We *could* write a combined task, but Rake gives us a better way—it lets us *compose* tasks. Here, for example, is a new task:

```
desc "Remove Unix and Windows backup files"
task :delete_backups =>
        [ :delete_unix_backups, :delete_windows_backups ] do
  puts "All backups deleted"
end
```

The task's name is delete_backups, and it depends on two other tasks. This isn't some special Rake syntax: we're simply passing the task method a Ruby hash containing a single entry whose key is the task name and whose value is the list of antecedent tasks. What this means is that Rake will execute the two platform-specific tasks before executing the delete_backups task:

```
% rake delete_backups
(in /Users/dave/OldWork/BS2/titles/RUBY3/Book/code/rake)
rm entry~
rm index.bak list.bak
All backups deleted
```

Our current Rakefile contains some duplication between the Unix and Windows deletion tasks. As it is just Ruby code, we can simply define a Ruby method to eliminate this:

```
def delete(pattern)
  files = Dir[pattern]
  rm(files, verbose: true) unless files.empty?
end
desc "Remove files whose names end with a tilde"
task :delete_unix_backups do
  delete "*~"
end
desc "Remove files with a .bak extension"
task :delete_windows_backups do
  delete "*.bak"
end
desc "Remove Unix and Windows backup files"
task :delete_backups => [ :delete_unix_backups, :delete_windows_backups ] do
  puts "All backups deleted"
end
```

If a Rake task is named default, it will be executed if you invoke Rake with no parameters.

You can find the tasks implemented by a Rakefile (or, more accurately, the tasks for which there is a description) using this:

```
% rake -T
(in /Users/dave/BS2/titles/RUBY3/Book/code/rake)
rake delete_backups          # Remove Unix and Windows backup files
rake delete_unix_backups     # Remove files whose names end with a tilde
rake delete_windows_backups  # Remove files with a .bak extension
```

This section only touches on the full power of Rake. It can handle dependencies between files (for example, rebuilding an executable file if one of the source files has changed), it knows about running tests and generating documentation, and it can even package gems for you. Martin Fowler has written a good overview of Rake if you're interested in digging deeper.[5] You might also want to investigate Sake, a tool that makes Rake tasks available no matter what directory you're in.[6]

Build Environment

When Ruby is compiled for a particular architecture, all the relevant settings used to build it (including the architecture of the machine on which it was compiled, compiler options, source code directory, and so on) are written to the module Config within the library file rbconfig.rb. After installation, any Ruby program can use this module to get details on how Ruby was compiled:

```
require 'rbconfig'
include Config
CONFIG["host"]     # =>   "i386-apple-darwin9.6.0"
CONFIG["libdir"]   # =>   "/usr/local/rubybook/lib"
```

Extension libraries use this configuration file in order to compile and link properly on any given architecture. See Chapter 29 beginning on page 833 and the reference for mkmf beginning on page 874 for details.

5. http://martinfowler.com/articles/rake.html

6. http://errtheblog.com/posts/60-sake-bomb