Origin

You may be interested to know that this document was originally written for internal use in the Operations department at Google. At the time, I was campaigning for the right to use Ruby on internal projects and I felt that a style and usage guide would probably assist in the battle for the language's acceptance, as the officially sanctioned languages at the time already had one. If nothing else, we'd at least all end up writing code that was easier to maintain.

Prologue

Over the last few years, Ruby has struck a chord with programmers as an excellent tool for -- amongst other things -- system administration. With that as our perspective, this document will suggest some guidelines for writing Ruby code in such a way that a common stylistic vocabulary may emerge, thereby increasing the ease with which source code may be read and shared.

If you have questions about this document, please address them to its author, Ian Macdonald.

Table of Contents

Tips for Productive Ruby Programming

An Admonishment

As you read this guide, try to keep in mind that the primary objective is to help you write legible code. Source code is a write-once/read-many-times medium, so don't be tempted to explore how much you can cram onto a single line or how you can employ side-effects in new and interesting ways. Leave the ostentation to the Perl world.

And even if you **are** cleverer than the rest of us, that's all the more reason to write code with us lesser mortals in mind, because you'll probably find yourself working alongside us one day. The good news is that Ruby itself goes a long way towards helping you write clear code; the rest is up to you.

irb

irb stands for Interactive Ruby. It gives you an interactive environment for trying out snippets of code and discovering what works and what doesn't. Its use can greatly decrease the amount of time you spend in the edit-run-debug cycle.

Additionally, it's a great way to familiarise yourself with the language, as it offers features such as method name completion and automatic printing of the result of each line or block of evaluated code. Getting used to using **irb** as a standard part of your development cycle and take advantage of its many features.

As a quick demonstration of the power of **irb**, we see in the following code which methods can be called with a **Fixnum** object (which is basically an integer) as the receiver:

```
[ianmacd@baghdad]$ irb
irb(main):001:0> 42.<Tab>
                          42.modulo
                          42.__id__                 42.next
                          42.__send__               42.nil?
                          42.abs                    42.nonzero?
                          42.between?               42.object_id
                          42.ceil                   42.prec
                          42.chr                    42.prec_f
                          42.class                  42.prec_i
                          42.clone                  42.private_methods
                          42.coerce                 42.protected_methods
                          42.display                42.public_methods
                          42.div                    42.quo
                          42.divmod                 42.remainder
                          42.downto                 42.respond_to?
                          42.dup                    42.round
                          42.eql?                   42.send
                          42.equal?                 42.singleton_method_added
                          42.extend                 42.singleton_methods
                          42.floor                  42.size
                          42.freeze                 42.step
                          42.frozen?                42.succ
                          42.hash                   42.taint
                          42.id                     42.tainted?
                          42.id2name                42.times
                          42.inspect                42.to_a
                          42.instance_eval          42.to_f
                          42.instance_of?           42.to_i
                          42.instance_variable_get  42.to_int
                          42.instance_variable_set  42.to_s
                          42.instance_variables     42.to_sym
                          42.integer?               42.truncate
                          42.is_a?                  42.type
                          42.kind_of?               42.untaint
                          42.method                 42.upto
                          42.methods                42.zero?
```

Similarly, "foo".<Tab> would show you a list of all methods that you can invoke on an object of class **String**:

```
irb(main):001:0> "foo".<Tab>
                      .gsub!                    .rjust
.__id__               .hash                     .rstrip
.__send__             .hex                      .rstrip!
.all?                 .id                       .scan
.any?                 .include?                 .select
.between?             .index                    .send
.capitalize           .inject                   .singleton_methods
.capitalize!          .insert                   .size
.casecmp              .inspect                  .slice
.center               .instance_eval            .slice!
```

```
.chomp                  .instance_of?           .sort
.chomp!                 .instance_variable_get  .sort_by
.chop                   .instance_variable_set  .split
.chop!                  .instance_variables     .squeeze
.class                  .intern                 .squeeze!
.clone                  .is_a?                  .strip
.collect                .kind_of?               .strip!
.concat                 .length                 .sub
.count                  .ljust                  .sub!
.crypt                  .lstrip                 .succ
.delete                 .lstrip!                .succ!
.delete!                .map                    .sum
.detect                 .match                  .swapcase
.display                .max                    .swapcase!
.downcase               .member?                .taint
.downcase!              .method                 .tainted?
.dump                   .methods                .to_a
.dup                    .min                    .to_f
.each                   .next                   .to_i
.each_byte              .next!                  .to_s
.each_line              .nil?                   .to_str
.each_with_index        .object_id              .to_sym
.empty?                 .oct                    .tr
.entries                .partition              .tr!
.eql?                   .private_methods        .tr_s
.equal?                 .protected_methods      .tr_s!
.extend                 .public_methods         .type
.find                   .reject                 .unpack
.find_all               .replace                .untaint
.freeze                 .respond_to?            .upcase
.frozen?                .reverse                .upcase!
.grep                   .reverse!               .upto
.gsub                   .rindex                 .zip
```

This works for any object and, since in Ruby everything is an object, even classes and modules, it means you can discover how to manipulate the entire object space and try out anything you want.

If you find that completion doesn't work for you, try starting **irb** as follows:

```
irb --readline -r irb/completion
```

ri

ri is to Ruby what perldoc is to Perl. The name stands for Ruby Interactive (not to be confused with Interactive Ruby).

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
```

On my system, that results in the following:

```
------------------------------------------------------------ Class: Hash
     A Hash is a collection of key-value pairs. It is similar to an
     Array, except that indexing is done via arbitrary keys of any
     object type, not an integer index. The order in which you traverse
     a hash by either key or value may seem arbitrary, and will
     generally not be in the insertion order.

     Hashes have a default value that is returned when accessing keys
     that do not exist in the hash. By default, that value is nil.

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

Includes:
     Enumerable(all?, any?, collect, detect, each_with_index, entries,
     find, find_all, grep, include?, inject, map, max, member?, min,
     partition, reject, select, sort, sort_by, to_a, to_set, zip)

Class methods:
     [], new

Instance methods:
     ==, [], []=, clear, default, default=, default_proc, delete,
     delete_if, each, each_key, each_pair, each_value, empty?, fetch,
     has_key?, has_value?, include?, index, indexes, indices,
     initialize_copy, inspect, invert, key?, keys, length, member?,
     merge, merge!, rehash, reject, reject!, replace, select, shift,
     size, sort, store, to_a, to_hash, to_s, update, value?, values,
     values_at
```

Next, try viewing the documentation for a given method. I'll pick one from the **Array** class, rather than **Hash**, just to give you as much variety as possible:

```
ri 'Array#<<'
```

This gives:

```
------------------------------------------------------------- Array#<<
     array << obj            => array
------------------------------------------------------------------------
     Append---Pushes the given object on to the end of this array. This
     expression returns the array itself, so several appends may be
     chained together.

       [ 1, 2 ] << "c" << "d" << [ 3, 4 ]
               #=>  [ 1, 2, "c", "d", [ 3, 4 ] ]
```

As you can see, 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.

It should be clear to you that **ri** is an unmissable tool in any Ruby programmer's toolbox.

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.

We won't go into the debugger in detail here. We simply wanted to make you aware of its existence, as no programming toolkit is complete without one. Dave Thomas has written extensive documentation on the subject.

Benchmarking

Ruby comes with a benchmarking module that can help you determine the fastest approach to tackling a given problem. The following snippet of code, again an example by Dave Thomas, shows the benchmarking of two loops. One declares a variable once outside of the loop, so that it persists when the loop ends. The other assigns to the variable within the loop, so that it is created local to the block on each iteration.

```
require "benchmark"
include Benchmark

n = 1000000
bm(12) do |test|
  test.report("normal:")    do
    n.times do |x|
      y = x + 1
    end
```

```
  end
  test.report("predefine:") do
    x = y = 0
    n.times do |x|
      y = x + 1
    end
  end
end
```

When run, the following output is generated:

```
                  user     system      total        real
normal:       0.920000   0.000000   0.920000 (  0.978238)
predefine:    0.720000   0.000000   0.720000 (  0.761112)
```

As you can see, it's a good 20% faster to predefine *y* outside of the iterator block, rather than create a block-local *y* variable on each iteration, which must then also be garbage-collected.

Running your code through the benchmarking module will slow it down tremendously, but the experience can be very worthwhile.

Profiling

Related to benchmarking is the subject of profiling. It probably won't surprise you by now to learn that Ruby also has a profiling module. Here's some sample code, once again courtesy of Dave Thomas:

```
require "profile"
class Peter
  def initialize(amt)
    @value = amt
  end


  def rob(amt)
    @value -= amt
    amt
  end
end

class Paul
  def initialize
    @value = 0
  end
```

```
  def pay(amt)
    @value += amt
    amt
  end
end

peter = Peter.new(1000)
paul = Paul.new
1000.times do
  paul.pay(peter.rob(10))
end
```

The output of this is below.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 45.83     0.11      0.11     1000     0.11     0.12  Peter#rob
 25.00     0.17      0.06        1    60.00   240.00  Integer#times
 20.83     0.22      0.05     1000     0.05     0.06  Paul#pay
  4.17     0.23      0.01     1000     0.01     0.01  Fixnum#-
  4.17     0.24      0.01     1000     0.01     0.01  Fixnum#+
  0.00     0.24      0.00        1     0.00     0.00  Paul#initialize
  0.00     0.24      0.00        4     0.00     0.00  Module#method_added
  0.00     0.24      0.00        2     0.00     0.00  Class#inherited
  0.00     0.24      0.00        1     0.00     0.00  Peter#initialize
  0.00     0.24      0.00        1     0.00   240.00  #toplevel
  0.00     0.24      0.00        1     0.00     0.00  Profiler__.start_profile
  0.00     0.24      0.00        2     0.00     0.00  Class#new
```

You don't even have to add the require 'profile' line to your program. You can just run ruby -rprofile my_program to get the same effect.

Incidentally, the profiling module in profiler.rb consists of just 59 lines of Ruby. Not bad for such a functional module!

Writing Ruby Unit Tests

Ruby includes the Test::Unit module specifically for the purpose of writing unit tests.

Using this library is criminally simple; so much so, in fact, that it would be a crime to write any more code that does not use unit tests to verify its correct working. Here are the steps to creating and using unit tests in Ruby:

- Require 'test/unit' in your test script.
- Create a class that subclasses Test::Unit::TestCase.
- Add a method that begins with "test" to your class.
- Make assertions in your test method.

- Optionally define #setup and/or #teardown to set up and/or tear down your common test fixture.

Yes, it really is **that** easy.

Here's an example from the Ruby/Password library's unit tests.

```
require 'test/unit'
require 'password'

class TC_PasswordTest < Test::Unit::TestCase

  def test_check
    # Check for a weak password.
    pw = Password.new( 'foo' )
    assert_raises( Password::WeakPassword ) { pw.check }

    # Check for a good password.
    pw = Password.new( 'G@7flAxg' )
    assert_nothing_raised { pw.check }

    # Check for an exception on bad dictionary path.
    assert_raises( Password::DictionaryError ) { pw.check( '/tmp/nodict' ) }
  end

end
```

The above code checks the strength of a simplistic password, *foo*, to ensure that a Password::WeakPassword exception is raised. Then, it checks that a much more reasonable password, *G@7flAxg* does **not** raise an exception. Finally, a third check is performed to ensure that passing the Password#check method a path to a non-existent dictionary yields a Password::DictionaryError exception. Each individual test within a test method is referred to as an assertion and you should write as many of them as you can devise. You really can't ever have too many, when it comes to unit tests.

When this is run, output similar to that below is generated if no tests fail:

```
$ ./tc_password.rb
Loaded suite ./tc_password
Started
.........
Finished in 3.692666 seconds.

9 tests, 10007 assertions, 0 failures, 0 errors
```

As you can see, there are actually many more tests in this library's test suite than just the one shown above. In reality, you may find yourself writing more lines of unit test code than in the actual code you are testing! Whilst all this work may seem like overkill, it will yield dividends later when you come back several months from now to perform more work on your code. Things that you have long since forgotten were fragile and easily broken will no longer need to be manually checked after changes to the code. Just run your unit tests and hope no exceptions are raised.

Further Reading

Mailing-lists

- A number of other mailing-lists are run for and by the Ruby community, including the excellent **ruby-talk** list.

Paper Books

- Programming Ruby by Dave Thomas and Andy Hunt, of Pragmatic Programmer fame
- The Ruby Way by Hal Fulton
- Ruby In a Nutshell by Ruby's creator, Yukihiro 'Matz" Matsumoto
- Sams Teach Yourself Ruby in 21 Days by Mark Slagell

Electronic Books

- why's (poignant) guide to Ruby

Recommended Articles

- An introductory article by Dave Thomas
- An introductory article on Freshmeat

Recommended Tutorials

- Marc Slagell's excellent tutorial

Standard Resources

- The official site
- Ruby Central
- The Ruby Garden
- The Ruby FAQ

Advocacy

- 37 reasons to love Ruby

Vim

Vim comes with useful syntax and indentation files for Ruby. Editing Ruby code will be much faster and easier if you make use of these.

In addition, Red Hat Linux's Vim package has a Ruby interpreter compiled into it. This enables you to use Ruby as a text filter, passing a block of text through some arbitrary Ruby code and replacing it with the result. This is handy when you want to do something that is too difficult or even impossible using Vim's own features.

You can also use Ruby to access Vim internals. For example, the following code would set the height of the current window to 70 lines:

```
:ruby VIM::Window.current.height = 70
```

For more information, execute the following command in Vim:

```
:help ruby
```

Emacs

Ruby has an ruby-mode.el file that provides a Ruby mode for Emacs.

FIXME: an Emacs user needs to elaborate on this.

Ruby Style Guidelines

Programs are much easier to maintain when all of their components have a consistent style. The remainder of this document sets forth general stylistic guidelines that are commonly adhered to in the Ruby community. We ask that you, too, make an effort to adhere to them when writing Ruby code.

Organisation

Order your code as follows:

- header block with author's name, Perforce Id tag and a brief description of what the program or library is for.
- require statements
- include statements
- class and module definitions
- main program section
- testing code

Testing code would look something like this:

```
if __FILE__ == $0
  ...
end
```

That should look familiar to you if you have done any Python programming.

Warnings

We recommend that you invoke your scripts with the following shebang line, plus whichever extra command-line options are appropriate for your needs:

```
#!/usr/bin/ruby -w
```

This will ensure that you always run the same interpreter (do not use *env(1)*) and that any warnings generated by your code or the libraries you include will be displayed, so that you can decide how to proceed.

If the warnings get too noisy and you are certain they are not worth silencing by changing your code, you can turn off warnings for a section of your code by setting $VERBOSE to **nil**. Even better is to codify this in a method:

```
def silently(&block)
  warn_level = $VERBOSE
  $VERBOSE = nil
  result = block.call
  $VERBOSE = warn_level
  result
end
```

Since this method takes a block as its parameter, you can now pass it arbitrary chunks of code to execute without warnings:

```
silently { require 'net/https' }
```

Exceptions

Exceptions are a means of breaking out of the normal flow of control of a block of code in order to handle errors or other exceptional conditions. For those not used to working with them, they can initially seem awkward and difficult to anticipate. However, once you become accustomed to using them, you will find they reduce the need for a lot of explicit error-checking. This, in turn, helps reduce code-clutter. An important reason to use exceptions is that it's frequently better that a program not

continue after an unexpected condition or error arises, rather than continue to run with unknown data, perhaps with devastating consequences.

When you use exceptions in your own libraries, you may care to subclass one of the base exception classes for your own needs. This makes the exceptions more specific to your use and allows for later customisation.

Be specific about the exceptions you are catching in your rescue clauses. This helps avoid trapping errors of one kind and treating them as another, which might otherwise lead to misleading diagnostic messages and a longer debugging cycle than if the error hadn't been trapped at all.

Use the ensure clause to execute finalising code, regardless of whether an exception was raised in the preceding begin block. This is often useful for clean-up (e.g. closing a file), which is something that always needs to be done, whether or not an exception occurred during the handling of the file. Here's an example.

```
begin
file = open("/tmp/some_file", "w")
  # do stuff here
ensure
  file.close
end
```

Global variables

Global variables are occasionally useful for quick scripts that will never be reused. However, they can rapidly cause namespace conflicts and accidental references in larger programs. For this reason, global variables are pretty much completely shunned by the Ruby community and you are strongly advised to follow suit.

The sole exceptions to this are:

one-liners
: commands issued at the command-line with ruby -e are not written to file, so by definition they will not be reused in any significant way. It's fine to use global variables, such as $_ here.

predefined global variables
: such as $., $! and $?. The use of these variables is necessary, although you are encouraged to require 'English' when you do so, in order to improve code readability.

Just to be clear, constants are an entirely different matter. These are global to the module or class in which they are defined, but they are fine to use. Re-use of constants will, in any case, generate a warning.

Boolean values

Ruby considers **nil** and **false** to be false, and all other values to be **true**. This is one of the major obvious differences from Perl, with which Ruby shares many superficial similarities. In Perl, **0**, the null string and **undef** are all considered false for the purposes of Boolean logic. In Ruby, however, **0** and the null string both evaluate to true. Ruby also has the Boolean values of **true** and **false**, unlike Perl.

If you're coming from a Perl background, the important thing to remember is that you will need more explicit tests for empty strings and zero values than you are used to, such as:

```
if foo.empty?
  ...
end
```

```
if foo.zero?
  ...
end
```

These can be rewritten in more Perl-like syntax, if that better suits your taste.

```
if foo == ""
  ...
end
```

```
if foo == 0
  ...
end
```

Default Parameter values

You can specify values for variables as part of a method's parameter list, e.g.

```
def foo(a, b=0, c="zip")
  ...
end
```

If foo is called with only one argument, *b* is set to **0** and *c* is set to **zip**. If it is called with two arguments, *b* has the value of the second argument, while *c* will still be set to **zip**.

Default parameter values are particularly useful when a method has lots of parameters, most of which only rarely need to have a different value. Using default parameters allows you to avoid having to specify values for every method call.

Initialisation

Ruby allows several ways to instantiate base objects. For example, each of the following pairs are equivalent:

```
foo = ""
foo = String.new
```

```
foo = []
foo = Array.new
```

```
foo = {}
foo = Hash.new
```

We really have no preference for which form you use. The first form is idiomatic enough across multiple languages that an experienced programmer in other languages will recognise and understand these declarations even if they have not seen Ruby before. However, the second form is arguably clearer still, as even a non-programmer would understand these declarations.

An important fact to bear in mind is that the new method of many of these classes sometimes takes one or more parameters, sometimes even a block. This allows you to determine in advance, for example, which hash values will be automatically created the first time their keys are referenced. For more information, use ri to view the documentation for the Array.new and Hash.new methods.

In-line Evaluation

Ruby is an extremely dynamic language, which means that code can be generated and executed on the fly. This includes the definition of new classes and methods, as well as the overloading of operators. Existing classes can even be reopened and have methods added, removed and redefined.

This allows for extremely flexible and efficient coding. For example, a small loop can generate dozens of similar methods, rather than the programmer having to define each one by hand. Imagine, for example, a SOAP library that reads a WSDL file and generates methods for each of the remote procedures it defines. In fact, Ruby's own SOAP4R library works in exactly this way.

Be wary of this kind of technique. Whilst powerful, it can make bugs very hard to track down and test. Because code generation is dynamic, unit tests will also be unable to test the entire code. As such, we recommend you use dynamic method generation sparingly and only where it makes perfect sense (as in the SOAP example).

Similarly, avoid Kernel#eval where possible and resort to Object#send, Module#class_eval, Module#module_eval and Object#instance_eval where possible. These methods, whilst less dangerous, are also best avoided where an alternative exists.

Semi-colons

Ruby allows the use of the semi-colon as a statement separator. This means you can place one at the end of a line and also use it as a way to put multiple statements on a single line. Don't do this. A semi-colon at the end of the line is superfluous and placing multiple statements on one line decreases the legibility of your code.

A rare exception is when writing a compound statement, coupled with a statement modifier:

```
($stderr.puts "Error!"; exit 1) unless x == 2
```

But even then, it's rather ugly.

Another minor variation on this theme is variable assignment. Instead of this:

```
level = 0
size = 0
```

it's possible to write this:

```
level = size = 0
```

Be careful with this, however, as most values in Ruby are references. The above example works, because **Fixnum** objects are *immediate values*, which means that an assignment of a **Fixnum** to a variable causes that variable to be assigned the value itself, rather than

just a reference to it. The same is not true of any other type of object, except **Bignum** objects, **true**, **false** and **nil**.

For example:

```
irb(main):001:0> foo = bar = Array.new
=> []
irb(main):002:0> foo
=> []
irb(main):003:0> bar
=> []
irb(main):004:0> foo << "baz"
=> ["baz"]
irb(main):005:0> foo
=> ["baz"]
irb(main):006:0> bar
=> ["baz"]
```

As you can see here, both *foo* and *bar* are clearly references to the same object.

Line length

The maximum length of any line of your code should be 80 characters (some would even say 79). The only exceptions should be complex regular expressions and other constructs that are difficult to break over multiple lines.

Make use of Ruby's implied line continuation. Basically, if Ruby can determine that a statement is syntactically incomplete, it will expect it to be continued on the next line:

```
puts "This line appears to end"
  if foo == bar
```

The above example confuses Ruby, since the first line is a valid statement in its own right. If, however, the if modifier were moved to the end of the first line, Ruby could then determine that there must be more code to come. Basically, the rule of thumb is that you should try to break a line after a comma or an operator.

If you can't imply that a line continues by using syntactically incomplete statements, terminate the line with a backslash, as you would if you were writing a shell script. Ruby will then continue parsing on the next line.

Indentation

Indent your code with two spaces per logical level. Really, it's that simple. Unlike in some other communities, there is no contention in the Ruby community over whether to use two, four or eight spaces. Everyone is happy to use just two. Furthermore, never use tabs, which includes the practice of mixing tabs with spaces.

Blank Lines

We recommend two blank lines between each class and module definition, and a single blank line between each method definition.

Whitespace

Use whitespace around the assignment operator, except then assigning default values in method argument lists, i.e.:

```
foo = 1
```

not:

```
foo=1
```

and:

```
def foo(bar, baz=0)
```

not:

```
def foo(bar, baz = 0)
```

Use whitespace to improve the legibility of complex expressions, but do not use whitespace to separate a method name from its parameter list. Within the parameter list itself, however, whitespace is fine.

For example, this is fine:

```
foo.bar( baz )
```

but this is not:

```
foo.bar (baz)
```

One last place where whitespace is not appropriate is before a comma.

Comments

Document your libraries using RDoc. Rdoc was added to the base Ruby distribution as of 1.8.1 and is capable of producing HTML and ri documentation from your source code. As such, it is very powerful and you should spend a little time learning it. It's really quite simple once you get used to it and one of the nice things about it is that your code really does become self-documenting.

Leaving a blank comment line before a class, module or method definition makes the comment extra readable, e.g.:

```
# This method returns the answer to the universe.
#
def answer
  42
end
```

RD is on the way out as a form of documentation these days, but can still be useful for making the Ruby interpreter ignore a header or footer in your code, e.g.:

```
=begin
  * Name:
  * Description
  * Author:
  * Date:
  * License:
=end
```

All standard rules of good code commenting practice apply here, too. In other words, don't comment the blatantly obvious, such as i = i + 1, but do otherwise include lucid comments thoughout your code. Annotate particularly gruesome or unimplemented sections with **FIXME** and an explanation of what work remains to be done.

Strings

Use either single or double quotes around your strings unless you need to embed one type of quote in the string itself, in which case you should use the other type. If you require variable interpolation to take place, single quotes will not work, e.g.:

```
foo = "bar"
x = 'foo = #{foo}'  => "foo = #{foo}"
x = "foo = #{foo}"  => "foo = bar"
```

Consider using alternative forms of interpolation, however:

```
x = sprintf("foo = %s", foo)
x = "foo = %s" % [foo]
x = %Q(works like a "double-quoted" string and embedded " characters work fine)
x = %q(a 'single'-quoted string with embedded ' characters)
x = %w[an array of words]
```

Also, onsider the use of so-called *here documents* for multi-line strings:

```
print <<EOF
Usage: foo <bar>

Call foo with bar as an argument
EOF
```

Binding

Be aware of how tightly operators bind. This is an issue in Ruby, as it is in Perl and many other languages. For example:

```
puts nil || "foo"   # parses to: puts( nil || "foo" )  => prints "foo"
puts nil or "foo"   # parses to: puts( nil ) || "foo"  => prints nil
```

Block Style

Ruby allows the construction of iterator blocks using either do and end or braces ({}). Common practice dictates sticking to do and end, except when the entire iterator block fits on a single line. For example:

```
foo.each do |x|
  puts x
```

```
  x *= 2
  puts x
end
```

but:

```
large = foo.find_all { |x| x > 10 }
```

Libraries

As of this writing, Ruby has multiple libraries for achieving some similar ends. For example, **getopts** and **parsearg** are both libraries for command-line option parsing, but there is also **getoptlong** and the even the newer **optparse** library.

In general, when more than one library exists to fulfill a specific need, you should use the newest of the set. Not only will the newer libraries almost certainly be better than the ones they replace, but the next version of Ruby will likely deprecate the use of some of the older libraries and issue warnings when they are loaded. Eventually, such libraries will be removed, so plan ahead and use the newer libraries in your code.

In addition to **getopts** and **parsearg**, the following libraries should no longer be used in new code:

Deprecated	Use instead
cgi-lib	cgi
importenv	**none**
ftools	fileutils

Access Control

Access to instance variables in Ruby happens via explicitly declared reader, writer and accessor methods. There can be no accidental access. Avoid Object#instance_variable_set and Object#instance_variable_get, unless you're doing something very clever and you're sure it's justified. If in doubt, ask one of your Ruby-programming colleagues.

Designate your methods private or protected, as appropriate.

Naming

Avoid single character variable names for anything except counters and iterators.

The standard Ruby file extension is .rb, although many people working on UNIX-like systems don't bother with it for stand-alone scripts. Whether or not you use it for scripts is up to you, but you will need to use it for library files or they will not be found by the interpreter.

Class and module names should be in camel-case, e.g:

```
Class BigFatObject
```

Class names should be nouns. In the case of modules, it's harder to make a clear recommendation. The names of mix-ins (which are just modules) should, however, probably be adjectives, such as the standard **Enumerable** and **Comparable** modules.

Constants should be named using all upper-case characters and underscores, e.g.

```
BigFatObject::MAX_SIZE
```

They should also be nouns.

Method names should be named using all lower-case characters and underscores. If possible, the name should be a verb, e.g.

```
BigFatObject#pare_down
```

Variable names should be named using all lower-case characters and underscores, but unlike method names, try to pick nouns, e.g. size. This applies to local, global, instance and class variables alike. However, there's a case to be made for keeping the names of local variables shorter than those of variables that will be made externally available, such as instance variables. For example, you might want to use resp_code as a local variable, but @response_code as an instance variable.

Parentheses

Ruby allows you to omit the parentheses between a method name and its parameter list. In the majority of cases, you should not do this and a warning will be generated by the interpreter if you do. However, certain methods are commonly used without parentheses, such as Kernel#require, Module#include, Kernel#p and the attr_* methods.

Safe Levels

If you've done any Perl programming, you'll know the importance of taint-checking when it comes to handling externally derived data, such as environment variables or the form fields returned by an HTTP POST operation from an HTML form.

Ruby also has the concept of safe modes of operation, but whereas Perl offers just one, Ruby offers no fewer than four modes of heightened security, each offering incremental improvements over the previous. Thanks to Dave Thomas for the descriptions below.

Level	Description
0	No checking of the use of externally supplied (tainted) data is performed. This is Ruby's default mode.
>= 1	Ruby disallows the use of tainted data by potentially dangerous operations.
>= 2	Ruby prohibits the loading of program files from globally writable locations.
>= 3	All newly created objects are considered tainted.
>= 4	Ruby effectively partitions the running program in two. Nontainted objects may not be modified. Typically, this will be used to create a sandbox: the program sets up an environment using a lower $SAFE level, then resets $SAFE to 4 to prevent subsequent changes to that environment.

We strongly encourage you to use safe levels whenever there is any doubt as to the trustworthiness of the data you are handling. In fact, we're paranoid enought to suggest that you raise the safe level of your program even when you **are** handling trusted data. Doing so will make a future editor of your code think twice about adding insecure operations.

A full explanation of safe levels is beyond the scope of this document, so we refer you instead to this thorough treatment.

External Commands

Related to the previous point is the question of when to call external commands. Ruby, like Perl, is an extremely useful glue language, which means that if you've got some software that looks like a square peg and some other software that looks like a round hole, Ruby acts like a lathe (or perhaps a very large hammer, depending on your approach) to make the peg fit the hole.

Anyway, when programming, it's common to want to call external commands and either format or make decisions based on their output. This is dangerous for a number of reasons. Firstly, you can never be certain that you are invoking the binary you think you are. This is a security risk. Secondly, binaries get upgraded from time to time and the output you can correctly parse today may not be the output you get presented with tomorrow. This is a robustness risk. Thirdly, your code may need to run on multiple platforms. Calling binaries on UNIX-like systems will not work on Windows systems. This is a portability risk.

Therefore, when faced with a choice between calling an external binary or spending a little extra time creating a pure Ruby solution, take the latter route. For example, don't call stty to switch off terminal echo while you accept a password from the

keyboard. Instead, use the Ruby/Password library. Similarly, if you need to perform a search of an LDAP directory, don't call the ldapsearch utility directly; use the Ruby/LDAP library instead.

Consistency

Above all, be consistent. If you're editing code written by someone else, take a moment to look at the surrounding code and determine its style. If the previous author has deviated slightly from the guidelines set forth in this document, you should, too. Retaining consistency at the local file level is a factor that overrides the guidelines in this document. If you observe widespread violation of these guidelines, however, consider fixing the whole file and having a word with the original author(s).

The point of having style guidelines is to have a common style of coding. Ruby gives you the vocabulary and the grammar, but it is a combination of your own skill and creativity, together with the established practices of the user community from which your programming style is derived.

Having a clear and uniform style allows people to concentrate on what you're saying, rather than how you're saying it.
