

Ruby 2.5 Information and Documentation

OCTOBER, 2018

wlharvey4

Published by:

wlharvey4

Address Line 1

Address Line 2

etc.

Email: wlharvey4@emac.com

URL: <http://www.example.com/>

Copyright © 2018

wlharvey4

All Rights Reserved.

The Ruby2.5 Information and Documentation program is copyright © 2018 by wlharvey4.
It is published under the conditions of the GNU General Public License, version 3.

This is Edition 0.2f of *Ruby 2.5 Information and Documentation*.

Table of Contents

Preface	1
Intended Audience	1
What Is Covered	1
Typographical Conventions	1
Acknowledgements	1
1 Introduction	2
2 Documentation	3
2.1 Installing Ruby	3
2.1.1 Package Management Systems	3
2.1.1.1 Homebrew (OS X)	3
2.1.2 Installers	3
2.1.2.1 <code>ruby-build</code>	4
2.1.2.2 <code>ruby-install</code>	4
2.1.3 Managers	4
2.1.3.1 <code>chruby</code>	4
2.1.3.2 <code>rbenv</code>	4
2.1.3.3 RVM (“Ruby Version Manager”)	4
2.1.3.4 <code>uru</code>	4
2.1.4 Building From Source	5
2.1.4.1 Releases Page	5
2.1.4.2 Branches Page	5
2.1.4.3 Ruby Issue Tracking System	6
2.2 Developing Ruby	7
2.3 Getting Started	8
2.3.1 Try Ruby!	8
2.3.2 Official FAQ	8
2.3.2.1 FAQ Iterators	8
2.3.2.2 FAQ Syntax	10
2.3.2.3 FAQ Methods	14
2.3.2.4 FAQ Classes and Modules	15
2.3.2.5 FAQ Built-In Libraries	15
2.3.2.6 FAQ Extension Library	16
2.3.2.7 FAQ Other Features	16
2.3.3 Ruby Koans	17
2.3.4 Whys (Poignant) Guide to Ruby	17
2.3.5 Ruby in Twenty Minutes	17
2.3.5.1 Interactive Ruby	17
2.3.5.2 Defining Methods	18
2.3.5.3 Altering Classes	20
2.3.5.4 Large Class Definition	21

2.3.5.5	Run MegaGreeter	24
2.3.6	Ruby from Other Languages	24
2.3.6.1	To Ruby From C and C++	24
2.3.6.2	To Ruby From Java	27
2.3.6.3	To Ruby From Perl	28
2.3.6.4	To Ruby From PHP	29
2.3.6.5	To Ruby From Python	30
2.3.7	Important Language Features	31
2.3.7.1	Pointers on Iteration	32
2.3.7.2	Everything has a value	32
2.3.7.3	Symbols are not lightweight Strings	32
2.3.7.4	Everything is an Object	33
2.3.7.5	Variable Constants	33
2.3.7.6	Naming conventions	33
2.3.7.7	Keyword arguments	33
2.3.7.8	The universal truth	34
2.3.7.9	Access modifiers are Methods	34
2.3.7.10	Method access	35
2.3.7.11	Classes are open	36
2.3.7.12	Funny method names	36
2.3.7.13	Singleton methods	36
2.3.7.14	Missing methods	37
2.3.7.15	Message passing, not function calls	37
2.3.7.16	Blocks are Objects	37
2.3.7.17	Operators are syntactic sugar	38
2.3.8	Learning Ruby	38
2.3.9	Ruby Essentials	38
2.3.10	Learn to Program	38
2.4	Manuals	38
2.4.1	Ruby User's Guide	38
2.4.1.1	On What Ruby Is	38
2.4.1.2	On Simple Examples	39
2.4.1.3	On Strings	41
2.4.1.4	On Puzzle Program	42
2.4.1.5	Regular Expressions	43
2.4.1.6	On Arrays And Hashes	45
2.4.1.7	On Control Structures	46
2.4.1.8	Ruby User's Guide On Iterators	49
2.4.1.9	On Object-Oriented Thinking	50
2.4.1.10	On Methods	51
2.4.1.11	On Classes	52
2.4.1.12	On Inheritance	53
2.4.1.13	On Redefinition of Methods	54
2.4.1.14	On Access Control	56
2.4.1.15	On Singleton Methods	57
2.4.1.16	On Modules	58
2.4.1.17	On Procedure Objects (Procs)	59
2.4.1.18	On Variables	60

2.4.1.19	On Global Variables	60
2.4.1.20	On Instance Variables	61
2.4.1.21	On Local Variables	61
2.4.1.22	On Class Constants	63
2.4.1.23	On Exception Processing and rescue	64
2.4.1.24	On Exception Processing And ensure	66
2.4.1.25	On Accessors	66
2.4.1.26	On Object Initialization	69
2.4.1.27	On Nuts And Bolts	70
2.4.2	Programming Ruby	72
2.5	Reference Documentation	74
2.6	Editors and IDEs	74
2.7	Further Reading	74
List of Tables		75
Appendix A Utility Programs		76
A.1	<code>eval.rb</code>	76
A.1.1	<code>eval.rb</code> Module Code	77
A.1.2	<code>eval.rb</code> Indentation Deltas Code	77
A.1.3	<code>eval.rb</code> Main Get Line Code	78
A.1.4	<code>eval.rb</code> Main Process Line Code	78
A.1.4.1	<code>eval.rb</code> If Not Line Code	78
A.1.4.2	<code>eval.rb</code> If Is Line Code	79
Appendix B The Makefile		81
B.1	Makefile Variable Definitions	81
B.2	Default Rule	81
B.3	TWJR Rules	81
B.4	Clean Rules	82
Appendix C Code Chunk Summaries		83
C.1	Source File Definitions	83
C.2	Code Chunk Definitions	83
C.3	Code Chunk References	84
Bibliography		86
Index		87
Program Index		93

Preface

Text here.

Intended Audience

Text here.

What Is Covered

Text and chapter by chapter description here.

Typographical Conventions

This book is written in an enhanced version of **Texinfo**, the GNU documentation formatting language. A single Texinfo source file is used to produce both the printed and online versions of a program’s documentation. Because of this, the typographical conventions are slightly different than in other books you may have read.

Examples you would type at the command-line are preceded by the common shell primary and secondary prompts, ‘\$’ and ‘>’. Input that you type is shown *like this*. Output from the command is preceded by the glyph “`+`”. This typically represents the command’s standard output. Error messages, and other output on the command’s standard error, are preceded by the glyph “`error`”. For example:

```
$ echo hi on stdout
+ hi on stdout
$ echo hello on stderr 1>&2
error hello on stderr
```

In the text, command names appear in **this font**, while code segments appear in the same font and quoted, ‘*like this*’. Options look like this: **-f**. Some things are emphasized *like this*, and if a point needs to be made strongly, it is done **like this**. The first occurrence of a new term is usually its *definition* and appears in the same font as the previous occurrence of “definition” in this sentence. Finally, file names are indicated like this: `/path/to/our/file`.

Acknowledgements

1 Introduction

Ruby is . . .

A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.

2 Documentation

Here you will find pointers to manuals, tutorials and references that will come in handy when you feel like coding in Ruby.

2.1 Installing Ruby

Installation Methods

There are several ways to install Ruby:

- **Package Manager:** When you are on a UNIX-like operating system, using your systems package manager is the easiest way of getting started. However, the packaged Ruby version usually is not the newest one.
- **Installers:** can be used to install a specific or multiple Ruby versions. There is also an installer for Windows.
- **Managers** help you to switch between multiple Ruby installations on your system.
- **Source:** And finally, you can also build Ruby from source.

The following overview lists available installation methods for different needs and platforms.

2.1.1 Package Management Systems

If you cannot compile your own Ruby, and you do not want to use a third-party tool, you can use your systems package manager to install Ruby.

Certain members in the Ruby community feel very strongly that you should never use a package manager to install Ruby and that you should use tools instead. While the full list of pros and cons is outside of the scope of this page, the most basic reason is that most package managers have older versions of Ruby in their official repositories. If you would like to use the newest Ruby, make sure you use the correct package name, or use the tools described further below instead.

2.1.1.1 Homebrew (OS X)

Homebrew

On macOS (High) Sierra and OS X El Capitan, Ruby 2.0 is included.

Many people on OS X use Homebrew as a package manager. It is really easy to get a newer version of Ruby using Homebrew:

```
$ brew install ruby
```

This should install the latest Ruby version.

2.1.2 Installers

If the version of Ruby provided by your system or package manager is out of date, a newer one can be installed using a third-party installer. Some of them also allow you to install multiple versions on the same system; associated managers can help to switch between the different Rubies. If you are planning to use RVM as a version manager you do not need a separate installer, it comes with its own.

2.1.2.1 ruby-build

`ruby-build`

`rbenv`

`ruby-build` is a plugin for `rbenv` (see [Section 2.1.3.2 “`rbenv`”, page 4](#)), that allows you to compile and install different versions of Ruby into arbitrary directories. `ruby-build` can also be used as a standalone program without `rbenv`. It is available for OS X, Linux, and other UNIX-like operating systems.

2.1.2.2 ruby-install

`ruby-install` version manager `chruby` version switcher

`ruby-install`

`chruby`

`ruby-install` allows you to compile and install different versions of Ruby into arbitrary directories. There is also a sibling, `chruby` (see [Section 2.1.3.1 “`chruby`”, page 4](#)), which handles switching between Ruby versions. It is available for OS X, Linux, and other UNIX-like operating systems.

2.1.3 Managers

Many Rubyists use Ruby managers to manage multiple Rubies. They confer various advantages but are not officially supported. Their respective communities are very helpful, however.

2.1.3.1 chruby

`chruby` allows you to switch between multiple Rubies. `chruby` can manage Rubies installed by `ruby-install` (see [Section 2.1.2.2 “`ruby-install`”, page 4](#)) or even built from source.

2.1.3.2 rbenv

`rbenv`

`ruby-build`

`rbenv` allows you to manage multiple installations of Ruby. It does not support installing Ruby, but there is a popular plugin named `ruby-build` (see [Section 2.1.2.1 “`ruby-build`”, page 4](#)) to install Ruby. Both tools are available for OS X, Linux, or other UNIX-like operating systems.

2.1.3.3 RVM (“Ruby Version Manager”)

`RVM`

RVM allows you to install and manage multiple installations of Ruby on your system. It can also manage different gemsets. It is available for OS X, Linux, or other UNIX-like operating systems.

2.1.3.4 uru

`Uru`

Uru is a lightweight, multi-platform command line tool that helps you to use multiple Rubies on OS X, Linux, or Windows systems.

2.1.4 Building From Source

Ruby 2.5.1

Ruby Github

Of course, you can install Ruby from source. Download and unpack a tarball, then just do this:

```
$ ./configure
$ make
$ sudo make install
```

By default, this will install Ruby into `/usr/local`. To change, pass the `--prefix=DIR` option to the `./configure` script.

Using the third-party tools or package managers might be a better idea, though, because the installed Ruby won't be managed by any tools.

Installing from the source code is a great solution for when you are comfortable enough with your platform and perhaps need specific settings for your environment. It's also a good solution in the event that there are no other premade packages for your platform.

2.1.4.1 Releases Page

Releases Page

For more information about specific releases, particularly older releases or previews, see the Releases page.

This page lists individual Ruby releases.

Ruby 2.5.1 Released

[ruby-2.1.5.tar.gz](#)

Posted by naruse on 28 Mar 2018

This release includes some bug fixes and some security fixes.

- CVE-2017-17742: HTTP response splitting in WEBrick
- CVE-2018-6914: Unintentional file and directory creation with directory traversal in `tempfile` and `tmpdir`
- CVE-2018-8777: DoS by large request in WEBrick
- CVE-2018-8778: Buffer under-read in `String#unpack`
- CVE-2018-8779: Unintentional socket creation by poisoned NUL byte in `UNIXServer` and `UNIXSocket`
- CVE-2018-8780: Unintentional directory traversal by poisoned NUL byte in `Dir`
- Multiple vulnerabilities in RubyGems

2.1.4.2 Branches Page

Branches Page

Information about the current maintenance status of the various Ruby branches can be found on the Branches page.

This page lists the current maintenance status of the various Ruby branches. This is a preliminary list of Ruby branches and their maintenance status. The shown dates are inferred from the English versions of release posts or EOL announcements.

The Ruby branches or release series are categorized below into the following phases:

- normal maintenance (bug fix): Branch receives general bug fixes and security fixes.
- security maintenance (security fix): Only security fixes are backported to this branch.
- eol (end-of-life): Branch is not supported by the ruby-core team any longer and does not receive any fixes. No further patch release will be released.
- preview: Only previews or release candidates have been released for this branch so far.

Ruby 2.6

<https://cache.ruby-lang.org/pub/ruby/2.6/ruby-2.6.0-preview2.tar.gz>

ruby-2.6.0-preview2

status: preview

release date:

Ruby 2.5

<https://cache.ruby-lang.org/pub/ruby/2.5/ruby-2.5.1.tar.gz>

status: normal maintenance

release date: 2017-12-25

Ruby 2.4

<https://cache.ruby-lang.org/pub/ruby/2.4/ruby-2.4.4.tar.gz>

status: normal maintenance

release date: 2016-12-25

Ruby 2.3

<https://cache.ruby-lang.org/pub/ruby/2.3/ruby-2.3.7.tar.gz>

status: security maintenance

release date: 2015-12-25

EOL date: scheduled for 2019-03-31

Ruby 2.2

status: eol

release date: 2014-12-25

EOL date: 2018-03-31

2.1.4.3 Ruby Issue Tracking System

Bugs

How to report a bug

How To Report

Ruby Trunk

[Ruby Trunk](#)

[All Issues](#)

2.2 Developing Ruby

[Ruby Core](#)

Now is a fantastic time to follow Rubys development. With the increased attention Ruby has received in the past few years, theres a growing need for good talent to help enhance Ruby and document its parts. So, where do you start?

Ruby Core

The topics related to Ruby development covered here are:

- [“Developing Ruby”](#), page 7,
- [“Developing Ruby”](#), page 7,
- [“Patch by Patch”](#), page 7,
- Rules for Core Developers

Using Subversion to Track Ruby Development

Getting the latest Ruby source code is a matter of an anonymous checkout from the [Subversion](#) repository. From your command line:

```
$ svn co https://svn.ruby-lang.org/repos/ruby/trunk ruby
```

The `ruby` directory will now contain the latest source code for the development version of Ruby (`ruby-trunk`). Currently patches applied to the trunk are backported to the stable 2.5, 2.4, and 2.3 branches (see below).

If youd like to follow patching of Ruby 2.5, you should use the `ruby_2_5` branch when checking out:

```
$ svn co https://svn.ruby-lang.org/repos/ruby/branches/ruby_2_5
```

This will check out the respective development tree into a `ruby_2_5` directory. Developers working on the maintenance branches are expected to migrate their changes to Rubys trunk, so often the branches are very similar, with the exception of improvements made by Matz and Nobu to the language itself.

If you prefer, you may browse [Rubys Subversion repository via the web](#).

How to Use Git With the Main Ruby Repository

Those who prefer to use Git over Subversion can find instructions with the [mirror on GitHub](#), both for those with commit access and [everybody else](#).

Improving Ruby, Patch by Patch

The core team maintains an [issue tracker](#) for submitting patches and bug reports to Matz and the gang. These reports also get submitted to the [Ruby-Core mailing list](#) for discussion, so you can be sure your request wont go unnoticed. You can also send your patches straight to the mailing list. Either way, you are encouraged to take part in the discussion that ensues.

Please look over the [Patch Writers Guide](#) for some tips, straight from Matz, on how to get your patches considered.

[Steps for Building a Patch](#)

2.3 Getting Started

2.3.1 Try Ruby!

[Try Ruby!](#)

An interactive tutorial that lets you try out Ruby right in your browser. This 15-minute tutorial is aimed at beginners who want to get a feeling of the language.

2.3.2 Official FAQ

The official frequently asked questions.

[FAQ](#)

This document contains Frequently Asked Questions about Ruby with answers.

This FAQ is based on [The Ruby Language FAQ](#) originally compiled by Shugo Maeda and translated into English by Kentaro Goto. Thanks to Zachary Scott and Marcus Stollsteimer for incorporating the FAQ into the site and for a major overhaul of the content.

- General questions
- How does Ruby stack up against?
- Installing Ruby
- Variables, constants, and arguments
- [Section 2.3.2.1 “FAQ Iterators”](#), page 8,
- [Section 2.3.2.2 “FAQ Syntax”](#), page 10,
- Methods
- Classes and modules
- Built-in libraries
- Extension library
- Other features

2.3.2.1 FAQ Iterators

What is an iterator?

An iterator is a method which accepts a block or a `Proc` object. In the source file, the block is placed immediately after the invocation of the method. Iterators are used to produce user-defined control structures — especially loops.

Lets look at an example to see how this works. Iterators are often used to repeat the same action on each element of a collection, like this:

```
data = [1, 2, 3]
data.each do |i|
  puts i
end
```

The `each` method of the array `data` is passed the `do ... end` block, and executes it repeatedly. On each call, the block is passed successive elements of the array.

You can define blocks with `{ ... }` in place of `do ... end`.

```
data = [1, 2, 3]
data.each { |i|
  puts i
}
```

This code has the same meaning as the last example. However, in some cases, precedence issues cause `do ... end` and `{ ... }` to act differently.

```
foobar a, b do ... end # foobar is the iterator.
foobar a, b { ... } # b is the iterator.
```

This is because `{ ... }` binds more tightly to the preceding expression than does a `do ... end` block. The first example is equivalent to ‘`foobar(a, b) do ... end`’, while the second is ‘`foobar(a, b { ... })`’.

How can I pass a block to an iterator?

You simply place the block after the iterator call. You can also pass a `Proc` object by prepending `&` to the variable or constant name that refers to the `Proc`.

How is a block used in an iterator?

This section or parts of it might be out-dated or in need of confirmation.

There are three ways to execute a block from an iterator method:

1. the `yield` control structure;

The `yield` statement calls the block, optionally passing it one or more arguments.

```
def my_iterator
  yield 1, 2
end
```

```
my_iterator {|a, b| puts a, b }
```

2. calling a `Proc` argument (made from a block) with `call`;

If a method definition has a block argument (the last formal parameter has an ampersand (`&`) prepended), it will receive the attached block, converted to a `Proc` object. This may be called using `proc.call(args)`.

```
def my_iterator(&b)
  b.call(1, 2)
end
```

```
my_iterator {|a, b| puts a, b }
```

and

3. using `Proc.new` followed by a `call`.

`Proc.new` (or the equivalent `proc` or `lambda` calls), when used in an iterator definition, takes the block which is given to the method as its argument and generates a procedure object from it. (`proc` and `lambda` are effectively synonyms.)

[Update needed: lambda behaves in a slightly different way and produces a warning ‘tried to create Proc object without a block’.]

```
def my_iterator
  Proc.new.call(3, 4)
  proc.call(5, 6)
  lambda.call(7, 8)
end
```

```
my_iterator {|a, b| puts a, b }
```

Perhaps surprisingly, `Proc.new` and friends do not in any sense consume the block attached to the method — each call to `Proc.new` generates a new procedure object out of the same block.

You can tell if there is a block associated with a method by calling `block_given?`.

What does `Proc.new` without a block do?

`Proc.new` without a block cannot generate a procedure object and an error occurs. In a method definition, however, `Proc.new` without a block implies the existence of a block at the time the method is called, and so no error will occur.

How can I run iterators in parallel?

See <http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/5252>

2.3.2.2 FAQ Syntax

List of FAQ items:

- “FAQ Syntax”, page 10,
- “FAQ Syntax”, page 11,
- “FAQ Syntax”, page 11,
- “FAQ Syntax”, page 11,
- “FAQ Syntax”, page 11,
- “FAQ Syntax”, page 12,
- “FAQ Syntax”, page 12,
- “FAQ Syntax”, page 12,
- “FAQ Syntax”, page 12,
- “FAQ Syntax”, page 13,
- “FAQ Syntax”, page 13,
- “FAQ Syntax”, page 13,
- “FAQ Syntax”, page 13,
- “FAQ Syntax”, page 13,
- “FAQ Syntax”, page 14,

What is the difference between an immediate value and a reference?

`Fixnum`, `true`, `nil`, and `false` are implemented as *immediate values*. With immediate values, variables hold the objects themselves, rather than references to them.

Singleton methods cannot be defined for such objects. Two `Fixnums` of the same value always represent the same object instance, so (for example) instance variables for the `Fixnum` with the value 1 are shared between all the 1's in the system. This makes it impossible to define a singleton method for just one of these.

What is the difference between `nil` and `false`?

First the similarity: `nil` and `false` are the only two objects that evaluate to `false` in a boolean context. (In other words: they are the only “falsy” values; all other objects are “truthy”.)

However, `nil` and `false` are instances of different classes (`NilClass` and `FalseClass`), and have different behavior elsewhere.

We recommend that *predicate methods* (those whose name ends with a question mark) return `true` or `false`. Other methods that need to indicate failure should return `nil`.

The Empty String

An empty string (`""`) returns `true` in a conditional expression! In Perl, it's `false`. Its very simple: in Ruby, only `nil` and `false` are `false` in conditional contexts.

You can use `empty?`, compare the string to `""`, or compare the string's size or length to 0 to find out if a string is empty.

A Symbol Object

What does `:name` mean?

A colon followed by a name generates a *Symbol object* which corresponds one-to-one with the identifier. During the duration of a program's execution the same Symbol object will be created for a given name or string. Symbols can also be created with `"name".intern` or `"name".to_sym`.

Symbol objects can represent identifiers for methods, variables, and so on. Some methods, like `define_method`, `method_missing`, or `trace_var`, require a symbol. Other methods, e.g. `attr_accessor`, `send`, or `autoload`, also accept a string.

Due to the fact that they are created only once, Symbols are often used as hash keys. String hash keys would create a new object for every single use, thereby causing some memory overhead. There is even a special syntax for symbol hash keys:

```
person_1 = { :name => "John", :age => 42 }
person_2 = { name: "Jane", age: 24 }      # alternate syntax
```

Symbols can also be used as enumeration values or to assign unique values to constants:

```
status = :open # :closed, ...
```

```
NORTH = :NORTH
SOUTH = :SOUTH
```


How can I access the value of a symbol?

To get the value of the variable corresponding to a symbol, you can use `symbol.to_s` or `"#{symbol}"` to get the name of the variable, and then `eval` that in the scope of the symbol to get the variables contents:

```
a = "This is the content of 'a'"
b = eval("#{:a}")
a.object_id == b.object_id # => true
```

You can also use:

```
b = binding.local_variable_get(:a)
```

If your symbol corresponds to the name of a method, you can use `send`:

```
class Demo
  def hello
    "Hello, world"
  end
end
```

```
demo = Demo.new
demo.send(:hello)
```

Or you can use `Object#method` to return a corresponding `Method` object, which you may then call:

```
m = demo.method(:hello) # => #<Method: Demo#hello>
m.call                  # => "Hello, world"
```

Is loop a control structure?

Although `loop` looks like a control structure, it is actually a method defined in `Kernel`. The block which follows introduces a new scope for local variables.

Ruby doesnt have a post-test loop

Ruby does not have a `do { ... } while` construct, so how can I implement loops that test the condition at the end?

Clemens Hintze says: “You can use a combination of Rubys `begin ... end` and the `while` or `until` statement modifiers to achieve the same effect:

```
i = 0
begin
  puts "i = #{i}"
  i += 1
end until i > 4
```

Why cant I pass a hash literal to a method: p {}?

The `{}` is parsed as a block, not a `Hash` constructor. You can force the `{}` to be treated as an expression by making the fact that it's a parameter explicit: `p({})`.

I cant get `def pos=(val)` to work!

I have the following code, but I cannot use the method `pos = 1`.

```
def pos=(val)
  @pos = val
  puts @pos
end
```

Methods with `=` appended must be called with an explicit receiver (without the receiver, you are just assigning to a local variable). Invoke it as `self.pos = 1`.

What is the difference between `\1` and `\\1`?

They have the same meaning. In a single quoted string, only `\'` and `\\` are transformed and other combinations remain unchanged.

However, in a double quoted string, `"\1"` is the byte `\001` (an octal bit pattern), while `"\\1"` is the two character string containing a backslash and the character `"1"`.

What is the difference between `..` and `...`?

`..` includes the right hand side in the range, while `...` does not:

```
(5..8).to_a  # => [5, 6, 7, 8]
(5...8).to_a # => [5, 6, 7]
```

What is the difference between `or` and `||`?

`p(nil || "Hello")` prints `"Hello"`, while `p(nil or "Hello")` gives a parse error. Why?

`or` has a very low precedence; `p((nil or "Hello"))` will work.

The precedence of `or` is for instance also lower than that of `=`, whereas `||` has a higher precedence:

```
foo = nil || "Hello" # parsed as: foo = (nil || "Hello")
foo # => "Hello"
```

but perhaps surprisingly:

```
foo = nil or "Hello" # parsed as: (foo = nil) or "Hello"
foo # => nil
```

`or` (and similarly `and`) is best used, not for combining boolean expressions, but for control flow, like in:

```
do_something or raise "some error!"
```

where `do_something` returns `false` or `nil` when an error occurs.

Does Ruby have function pointers?

A `Proc` object generated by `Proc.new`, `proc`, or `lambda` can be referenced from a variable, so that variable could be said to be a function pointer. You can also get references to methods within a particular object instance using `object.method`.

What is the difference between load and require?

`load` will load and execute a Ruby program (*.rb).

`require` loads Ruby programs as well, but will also load *binary Ruby extension modules* (shared libraries or DLLs). In addition, `require` ensures that a feature is never loaded more than once.

Does Ruby have exception handling?

Ruby supports a flexible exception handling scheme:

```
begin
  statements which may raise exceptions
rescue [exception class names]
  statements when an exception occurred
rescue [exception class names]
  statements when an exception occurred
ensure
  statements that will always run
end
```

If an exception occurs in the `begin` clause, the `rescue` clause with the matching exception name is executed. The `ensure` clause is executed whether an exception occurred or not. `rescue` and `ensure` clauses may be omitted.

If no exception class is designated for a `rescue` clause, `StandardError` exception is implied, and exceptions which are in a `is_a?` relation to `StandardError` are captured.

This expression returns the value of the `begin` clause.

The latest exception is accessed by the global variable `$!` (and so its type can be determined using `$!.type`).

2.3.2.3 FAQ Methods

How does Ruby choose which method to invoke?

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

Where are ++ and -- ?

What is a singleton method?

All these objects are fine, but does Ruby have any simple functions?

So where do all these function-like methods come from?

Can I access an objects instance variables?

Whats the difference between private and protected?

How can I change the visibility of a method?

Can an identifier beginning with a capital letter be a method name?

Calling `super` gives an `ArgumentError`.

How can I call the method of the same name two levels up?

How can I invoke an original built-in method after redefining it?

What is a destructive method?

Why can destructive methods be dangerous?

Can I return multiple values from a method?

2.3.2.4 FAQ Classes and Modules

Can a class definition be repeated?

Are there class variables?

What is a class instance variable?

What is the difference between class variables and class instance variables?

Does Ruby have class methods?

What is a singleton class?

What is a module function?

What is the difference between a class and a module?

Can you subclass modules?

Give me an example of a mixin

Why are there two ways of defining class methods?

What is the difference between `include` and `extend`?

What does `self` mean?

2.3.2.5 FAQ Built-In Libraries

What does `instance_methods(false)` return?

How do random number seeds work?

I read a file and changed it, but the file on disk has not changed.

How can I process a file and update its contents?

I wrote a file, copied it, but the end of the copy seems to be lost.

How can I get the line number in the current input file?

How can I use `less` to display my programs output?

What happens to a `File` object which is no longer referenced?

I feel uneasy if I don't close a file.

How can I sort files by their modification time?

How can I count the frequency of words in a file?

How can I sort strings in alphabetical order?

How can I expand tabs to spaces?

How can I escape a backslash in a regular expression?

What is the difference between `sub` and `sub!`?

Where does `\Z` match?

What is the difference between `thread` and `fork`?

How can I use `Marshal`?

How can I use `trap`?

2.3.2.6 FAQ Extension Library

How can I use Ruby interactively?

Is there a debugger for Ruby?

How can I use a library written in C from Ruby?

Can I use `Tcl/Tk` in Ruby?

`Tk` won't work. Why?

Can I use `gtk+` or `xforms` interfaces in Ruby?

How can I do date arithmetic?

2.3.2.7 FAQ Other Features

What does `a ? b : c` mean?

How can I count the number of lines in a file?

What do `MatchData#begin` and `MatchData#end` return?

How can I sum the elements in an array?

How can I use continuations?

2.3.3 Ruby Koans

Ruby Koans

The Koans walk you along the path to enlightenment in order to learn Ruby. The goal is to learn the Ruby language, syntax, structure, and some common functions and libraries. We also teach you culture.

2.3.4 Whys (Poignant) Guide to Ruby

Why's Guide to Ruby

An unconventional but interesting book that will teach you Ruby through stories, wit, and comics. Originally created by *why the lucky stiff*, this guide remains a classic for Ruby learners.

2.3.5 Ruby in Twenty Minutes

Ruby in Twenty Minutes

A nice tutorial covering the basics of Ruby. From start to finish it shouldn't take you more than twenty minutes. It makes the assumption that you already have Ruby installed. (If you do not have Ruby on your computer install it before you get started.)

2.3.5.1 Interactive Ruby

Ruby comes with a program that will show the results of any Ruby statements you feed it. Playing with Ruby code in interactive sessions like this is a terrific way to learn the language.

Open up IRB (which stands for Interactive Ruby).

```
? irb
-| irb(main):001:0>

irb(main):001:0> "Hello World"
=> "Hello World"
-| irb(main):002:0>
```

The second line is just IRB's way of telling us the result of the last expression it evaluated. To print:

```
irb(main):002:0> puts "Hello World"
-| Hello World
=> nil
-| irb(main):003:0>
```

`puts` is the basic command to print something out in Ruby. But then what's the '`=> nil`' bit? That's the result of the expression. `puts` always returns `nil`, which is Ruby's absolutely-positively-nothing value.

2.3.5.2 Defining Methods

Define a method:

```
irb(main):010:0> def hi
irb(main):011:1> puts "Hello World!"
irb(main):012:1> end
=> :hi
```

The code '`def hi`' starts the definition of the method. The next line is the body of the method. Finally, the last line `end` tells Ruby we're done defining the method. Ruby's response `-> :hi` tells us that it knows we're done defining the method.

Try running that method a few times:

```
irb(main):013:0> hi
Hello World!
=> nil
irb(main):014:0> hi()
Hello World!
=> nil
```

If the method doesn't take parameters that's all you need. You can add empty parentheses if you'd like, but they're not needed.

Define Method with a Parameter

What if we want to say hello to one person, and not the whole world? Just redefine `hi` to take a name as a parameter.

```
irb(main):015:0> def hi(name)
irb(main):016:1> puts "Hello #{name}!"
irb(main):017:1> end
=> :hi
irb(main):018:0> hi("Matz")
Hello Matz!
=> nil
```

What's the `#{name}` bit? That's Ruby's way of inserting something into a string. The bit between the braces is turned into a string (if it isn't one already) and then substituted into the outer string at that point. You can also use this to make sure that someone's name is properly capitalized:

```
irb(main):019:0> def hi(name = "World")
irb(main):020:1> puts "Hello #{name.capitalize}!"
irb(main):021:1> end
=> :hi
irb(main):022:0> hi "chris"
Hello Chris!
=> nil
irb(main):023:0> hi
```

```
Hello World!  
=> nil
```

A couple of other tricks to spot here. One is that we're calling the method without parentheses again. If it's obvious what you're doing, the parentheses are optional. The other trick is the default parameter `World`. What this is saying is "If the name isn't supplied, use the default name of `"World"`".

Create a Class

What if we want a real greeter around, one that remembers your name and welcomes you and treats you always with respect. You might want to use an object for that. Let's create a Greeter class.

```
irb(main):024:0> class Greeter  
irb(main):025:1>   def initialize(name = "World")  
irb(main):026:2>     @name = name  
irb(main):027:2>   end  
irb(main):028:1>   def say_hi  
irb(main):029:2>     puts "Hi #{@name}!"  
irb(main):030:2>   end  
irb(main):031:1>   def say_bye  
irb(main):032:2>     puts "Bye #{@name}, come back soon."  
irb(main):033:2>   end  
irb(main):034:1> end  
=> :say_bye
```

The new keyword here is `class`. This defines a new class called `Greeter` and a bunch of methods for that class. Also notice `@name`. This is an instance variable, and is available to all the methods of the class. As you can see it's used by `say_hi` and `say_bye`.

Create an Object

Now let's create a greeter object and use it:

```
irb(main):035:0> greeter = Greeter.new("Pat")  
=> #<Greeter:0x16cac @name="Pat">  
irb(main):036:0> greeter.say_hi  
Hi Pat!  
=> nil  
irb(main):037:0> greeter.say_bye  
Bye Pat, come back soon.  
=> nil
```

Instance Variables

Instance variables are hidden away inside the object. They're not terribly hidden, you see them whenever you inspect the object, and there are other ways of accessing them, but Ruby uses the good object-oriented approach of keeping data sort-of hidden away.

So what methods do exist for Greeter objects?

```
'Object#instance_methods'  
irb(main):039:0> Greeter.instance_methods
```



```
=> [:say_hi, :say_bye, :instance_of?, :public_send,
    :instance_variable_get, :instance_variable_set,
    :instance_variable_defined?, :remove_instance_variable,
    :private_methods, :kind_of?, :instance_variables, :tap,
    :is_a?, :extend, :define_singleton_method, :to_enum,
    :enum_for, :<=>, :==, :=~, :!~, :eql?, :respond_to?,
    :freeze, :inspect, :display, :send, :object_id, :to_s,
    :method, :public_method, :singleton_method, :nil?, :hash,
    :class, :singleton_class, :clone, :dup, :itself, :taint,
    :tainted?, :untaint, :untrust, :trust, :untrusted?, :methods,
    :protected_methods, :frozen?, :public_methods, :singleton_methods,
    :!, :==, :!=, :__send__, :equal?, :instance_eval, :instance_exec, :__id__]
```

We only defined two methods. Whats going on here? Well this is all of the methods for Greeter objects, a complete list, including ones defined by ancestor classes. If we want to just list methods defined for Greeter we can tell it to not include ancestors by passing it the parameter false, meaning we dont want methods defined by ancestors.

```
'Object#instance_methods(false)'
irb(main):040:0> Greeter.instance_methods(false)
=> [:say_hi, :say_bye]
```

So lets see which methods our greeter object responds to:

```
'Object#respond_to?'
irb(main):041:0> greeter.respond_to?("name")
=> false
irb(main):042:0> greeter.respond_to?("say_hi")
=> true
irb(main):043:0> greeter.respond_to?("to_s")
=> true
```

So, it knows `say_hi`, and `to_s` (meaning convert something to a string, a method that's defined by default for every object), but it doesn't know `name`.

2.3.5.3 Altering Classes

But what if you want to be able to view or change the name? Ruby provides an easy way of providing access to an object's variables.

```
'attr_accessor :name'
irb(main):044:0> class Greeter
irb(main):045:1>   attr_accessor :name
irb(main):046:1> end
=> nil
```

In Ruby, you can open a class up again and modify it. The changes will be present in any new objects you create and even available in existing objects of that class. So, lets create a new object and play with its `@name` property.

```
irb(main):047:0> greeter = Greeter.new("Andy")
=> #<Greeter:0x3c9b0 @name="Andy">
```

```

irb(main):048:0> greeter.respond_to?("name")
=> true
irb(main):049:0> greeter.respond_to?("name=")
=> true
irb(main):050:0> greeter.say_hi
Hi Andy!
=> nil
irb(main):051:0> greeter.name="Betty"
=> "Betty"
irb(main):052:0> greeter
=> #<Greeter:0x3c9b0 @name="Betty">
irb(main):053:0> greeter.name
=> "Betty"
irb(main):054:0> greeter.say_hi
Hi Betty!
=> nil

```

Using `attr_accessor` defined two new methods for us, `name` to get the value, and `name=` to set it.

2.3.5.4 Large Class Definition

What if we had some kind of `MegaGreeter` that could either greet the world, one person, or a whole list of people? Lets write this one in a file instead of directly in the interactive Ruby interpreter IRB.

```

{ri20min.rb} ≡

#!/usr/bin/env ruby

class MegaGreeter
  attr_accessor :names

  <MegaGreeter—Initialize Method>
  <MegaGreeter—say_hi Method>
  <MegaGreeter—say_bye Method>
end

if __FILE__ == $0
  <MegaGreeter—Main Script>
end

```

The following table lists called chunk definition points.

Chunk name	First definition point
<MegaGreeter—Initialize Method>	See “Large Class Definition”, page 22.
<MegaGreeter—Main Script>	See “Large Class Definition”, page 23.
<MegaGreeter—say_bye Method>	See “Large Class Definition”, page 23.
<MegaGreeter—say_hi Method>	See “Large Class Definition”, page 22.

Initialize Method

<MegaGreeter—Initialize Method> ≡

```
# Create the object
def initialize(names = "World")
  @names = names
end
```

This chunk is called by {ri20min.rb}; see its first definition at “Large Class Definition”, page 21.

say_hi Method

The `say_hi` method has become a bit more complicated. It now looks at the `@names` instance variable to make decisions. If it’s `nil`, it just prints out three dots. No point greeting nobody, right?

If the `@names` object responds to `each`, it is something that you can iterate over, so iterate over it and greet each person in turn. Finally, if `@names` is anything else, just let it get turned into a string automatically and do the default greeting.

<MegaGreeter—say_hi Method> ≡

```
# Say hi to everybody
def say_hi
  if @names.nil?
    puts "..."
  elsif @names.respond_to?("each")
    # @names is a list of some kind, iterate!
    @names.each do |name|
      puts "Hello #{name}!"
    end
  else
    puts "Hello #{@names}!"
  end
end
```

This chunk is called by {ri20min.rb}; see its first definition at “Large Class Definition”, page 21.

The Iterator

Lets look at that iterator in more depth:

```
@names.each do |name|
  puts "Hello #{name}!"
end
```

`each` is a method that accepts a block of code then runs that block of code for every element in a list, and the bit between `do` and `end` is just such a block. A *block* is like an anonymous function or lambda. The variable between pipe characters is the parameter for this block.

What happens here is that for every entry in a list, `name` is bound to that list element, and then the expression `puts "Hello #{name}!"` is run with that name.

Internally, the `each` method will essentially call `yield "Albert"`, then `yield "Brenda"` and then `yield "Charles"`, and so on.

The Real Power of Blocks

The real power of blocks is when dealing with things that are more complicated than lists. Beyond handling simple housekeeping details within the method, you can also handle setup, teardown, and errors all hidden away from the cares of the user.

say_bye Method

The `say_bye` method doesn't use `each`; instead it checks to see if `@names` responds to the `join` method, and if so, uses it. Otherwise, it just prints out the variable as a string.

Duck Typing

This method of not caring about the actual type of a variable, just relying on what methods it supports is known as *Duck Typing*, as in “if it walks like a duck and quacks like a duck. . .”. The benefit of this is that it doesn't unnecessarily restrict the types of variables that are supported. If someone comes up with a new kind of list class, as long as it implements the `join` method with the same semantics as other lists, everything will work as planned.

<MegaGreeter—say_bye Method> ≡

```
# Say bye to everybody
def say_bye
  if @names.nil?
    puts "..."
  elsif @names.respond_to?("join")
    # Join the list elements with commas
    puts "Goodbye #{@names.join(", ")}. Come back soon!"
  else
    puts "Goodbye #{@names}. Come back soon!"
  end
end
```

This chunk is called by `{ri20min.rb}`; see its first definition at “[Large Class Definition](#)”, page 21.

MegaGreeter Main Script

There's one final trick to notice, and that's the line:

```
if __FILE__ == $0
```

`__FILE__` is the magic variable that contains the name of the current file. `$0` is the name of the file used to start the program. This check says “If this is the main file being used. . .” This allows a file to be used as a library, and not to execute code in that context, but if the file is being used as an executable, then execute that code.

<MegaGreeter—Main Script> ≡

```
mg = MegaGreeter.new
mg.say_hi
mg.say_bye
```

```
# Change name to be "Zeke"
mg.names = "Zeke"
mg.say_hi
mg.say_bye

# Change the name to an array of names
mg.names = ["Albert", "Brenda", "Charles",
            "Dave", "Engelbert"]

mg.say_hi
mg.say_bye

# Change to nil
mg.names = nil
mg.say_hi
mg.say_bye
```

This chunk is called by `{ri20min.rb}`; see its first definition at [“Large Class Definition”](#), page 21.

2.3.5.5 Run MegaGreeter

Run the program `ri20min.rb` as `‘ruby ri20min.rb’`. The output should be:

```
Hello World!
Goodbye World.  Come back soon!
Hello Zeke!
Goodbye Zeke.  Come back soon!
Hello Albert!
Hello Brenda!
Hello Charles!
Hello Dave!
Hello Engelbert!
Goodbye Albert, Brenda, Charles, Dave, Engelbert.  Come back soon!
...
...
```

2.3.6 Ruby from Other Languages

[Ruby from Other Languages](#)

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

2.3.6.1 To Ruby From C and C++

Everything Is Different

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

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

Ruby is Quicker to Code But Slower to Execute

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

Dynamically Typed

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

Extension Modules

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

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

Similarities With C

- You may program procedurally if you like (but it will still be object-oriented behind the scenes).
- Most of the operators are the same (including the compound assignment and also bitwise operators). Though, Ruby doesn’t have `++` or `--`.
- Ruby has `__FILE__` and `__LINE__`.
- You can also have constants, though there’s no special `const` keyword. `Const`-ness is enforced by a naming convention instead — names starting with a capital letter are for constants.
- Strings go in double-quotes and are mutable
- Just like man pages, you can read most docs in your terminal window — though using the `ri` command.
- You’ve got the same sort of command-line debugger available.

Similarities with C++

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

Differences From C

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

Differences from C++

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

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

2.3.6.2 To Ruby From Java

Ruby is Less Verbose

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

Similarities with Java

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

Differences From Java

- You don’t need to compile your code. You just run it directly.

- There are several different popular third-party GUI toolkits. Ruby users can try WxRuby, FXRuby, Ruby-GNOME2, Qt, or the bundled-in Ruby Tk for example.
- You use the `end` keyword after defining things like classes, instead of having to put braces around blocks of code.
- You have `require` instead of `import`.
- All member variables are private. From the outside, you access everything via methods.
- Parentheses in method calls are usually optional and often omitted.
- Everything is an object, including numbers like 2 and 3.14159.
- There's no static type checking.
- Variable names are just labels. They don't have a type associated with them.
- There are no type declarations. You just assign to new variable names as-needed and they just “spring up” (i.e. `a = [1,2,3]` rather than `int[] a = {1,2,3};`).
- There's no casting. Just call the methods. Your unit tests should tell you before you even run the code if you're going to see an exception.
- It's `foo = Foo.new("hi")` instead of `Foo foo = new Foo("hi")`.
- The constructor is always named `initialize` instead of the name of the class.
- You have “mixins” instead of interfaces.
- YAML tends to be favored over XML.
- It's `nil` instead of `null`.
- `==` and `equals()` are handled differently in Ruby. Use `==` when you want to test “equivalence” in Ruby (`equals()` in Java). Use `equal?()` when you want to know if two objects are “the same” (`==` in Java).

2.3.6.3 To Ruby From Perl

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

Similarities with Perl

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

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

Differences From Perl

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

2.3.6.4 To Ruby From PHP

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

Similarities with PHP

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

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

Differences From PHP

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

2.3.6.5 To Ruby From Python

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

Similarities With Python

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

Differences From Python

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

2.3.7 Important Language Features

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

2.3.7.1 Pointers on Iteration

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

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

For more info on `each` and its friends

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

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

2.3.7.2 Everything has a value

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

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

2.3.7.3 Symbols are not lightweight Strings

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

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

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

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

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

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

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

2.3.7.4 Everything is an Object

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

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

2.3.7.5 Variable Constants

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

2.3.7.6 Naming conventions

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

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

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

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

2.3.7.7 Keyword arguments

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

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

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

2.3.7.8 The universal truth

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

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

This will print ‘0 is false’. The equivalent Ruby:

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

Prints ‘0 is true’.

2.3.7.9 Access modifiers are Methods

Access modifiers apply until the end of scope.

In the following Ruby code,

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

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

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

  private

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

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

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

2.3.7.10 Method access

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

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

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

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

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

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

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

class Test
  protected :identifier
end

t1 == t2 # => true

# now make 'identifier' private

class Test
  private :identifier
end

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


2.3.7.11 Classes are open

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

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

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

2.3.7.12 Funny method names

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

2.3.7.13 Singleton methods

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

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

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

porsche.inspect # => Expensive car

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

2.3.7.14 Missing methods

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

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

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

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

2.3.7.15 Message passing, not function calls

A method call is really a “message” to another object:

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

2.3.7.16 Blocks are Objects

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

```
def block(&the_block)
  # Inside here, the_block is the block passed to the method
  the_block # return the block
end

adder = block { |a, b| a + b }
# adder is now a Proc object
adder.class # => Proc
```

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

Similarly, methods are also Objects in the making:

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

2.3.7.17 Operators are syntactic sugar

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

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

You don't need C++'s `operator+`, etc.

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

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

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

2.3.8 Learning Ruby

Learning Ruby

A thorough collection of Ruby study notes for those who are new to the language and in search of a solid introduction to Ruby's concepts and constructs.

2.3.9 Ruby Essentials

Ruby Essentials

2.3.10 Learn to Program

Learn to Program

A wonderful little tutorial by Chris Pine for programming newbies. If you don't know how to program, start here.

Learn Ruby the Hard Way

2.4 Manuals

2.4.1 Ruby User's Guide

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

Ruby User's Guide

2.4.1.1 On What Ruby Is

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

interpreted scripting language:

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

quick and easy:

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

object oriented programming:

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

also:

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

2.4.1.2 On Simple Examples

Factorial in Ruby

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

$$\begin{aligned} n! &= 1 && \text{(when } n=0\text{)} \\ &= n * (n-1)! && \text{(otherwise)} \end{aligned}$$

In ruby, this can be written as:

```
{fact.rb} ≡  
# Program to find the factorial of a number  
# Save this as fact.rb  
  
def fact(n)  
  if n == 0  
    1  
  else  
    n * fact(n-1)  
  end  
end  
  
puts fact(ARGV[0].to_i)
```

Command Line Arguments — In Array *ARGV*

ARGV is an array which contains the command line arguments, and `to_i` converts a character string to an integer.¹

The end Statement

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

Takeaway — return Statement Optional

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

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

Running `fact.rb`

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

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

% ruby fact.rb 40
815915283247897734345611269596115894272000000000

% ruby fact.rb 400
64034522846623895262347970319503005850702583026002959458684
44594280239716918683143627847864746326467629435057503585681
08482981628835174352289619886468029979373416541508381624264
61942352307046244325015114448670890662773914918117331955996
44070954967134529047702032243491121079759328079510154537266
72516278778900093497637657103263503315339653498683868313393
52024373788157786791506311858702618270169819740062983025308
59129834616227230455833952075961150530223608681043329725519
48526744322324386699484224042325998055516106359423769613992
31917134063858996537970147827206606320217379472010321356624
61380907794230459736069956759583609615871512991382228657857
95493616176544804532220078258184008484364155912294542753848
03558374518022675900061399560145595206127211192918105032491
008000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000
```

¹ Ruby does not convert strings into integers automatically like perl does.

The Input/Evaluation Loop

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

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

Ruby Evaluation Program — eval.rb

Ruby also comes with a program called `eval.rb` (see [Section A.1 “eval.rb”, page 76](#)) that allows you to enter ruby code from the keyboard in an interactive loop, showing you the results as you go. It will be used extensively through the rest of this guide. You should use this enhanced `eval.rb` that adds visual indenting assistance, warning reports, and color highlighting.

Here is a short `eval.rb` session:

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

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

No Distinction Between Statement and Expression

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

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

2.4.1.3 On Strings

Quoting Strings

A string may be double-quoted (“...”) or single-quoted (‘...’).

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

String Methods

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

Here are some things you can do with strings.

Concatenation

```
‘word = "fo" + "o" ⇒ “foo”
```

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

Extracting characters

(note that characters are integers in ruby)

`'word[0]' ⇒ '102' '# 102 is ASCII code of 'f''`

`'word[-1]' ⇒ '111' '# 111 is ASCII code of 'o''`

Extracting substrings

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

Testing for equality:

`"foo" == "foo" ⇒ 'true'`

`"foo" == "bar" ⇒ 'false'`

2.4.1.4 On Puzzle Program

`{guess.rb} ≡`

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

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

New Control Structure while

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

value of the whole ‘`guess=STDIN.gets`’ expression, has a `nil` value, causing `while` to stop looping).

Standard Input Object — *STDIN*

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

`guess.chop!`

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

Printing Variables

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

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

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

`print` vs `puts`

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

Flushing Standard Output

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

2.4.1.5 Regular Expressions

```
ruby> def chab(s)    # "contains hex in angle brackets"
  |      (s =~ /<(x|X)(\d|[a-f]|[A-F])+>/) != nil
  | end
  nil
ruby> chab "Not this one."
false
```



```

ruby> chab "Maybe this? {0x35}"      # wrong kind of brackets
      false
ruby> chab "Or this? <0x38z7e>"      # bogus hex digit
      false
ruby> chab "Okay, this: <0xfc0004>."
      true

```

Program To Help Experiment With Regular Expressions

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

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

```

{regx.rb} ≡
# Requires an ANSI terminal!

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

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

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

```

Explication of `regx.rb`

The `break` Statement

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

`if` Modifiers

```

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

```

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

situations where a statement and condition are short enough to fit comfortably together on one script line.

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

Nondestructive chops vs Destructive chops!

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

`chomp` And `chomp!`

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

Predicate Method Naming Convention

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

Regular Expressions At Work

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

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

The `=~` Matching Operator

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

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

2.4.1.6 On Arrays And Hashes

Creating An Array

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

Concatenating and Repeating Arrays

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

Referring To Elements of Arrays

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

Converted To And From Strings

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

Hashes

An *associative array* has elements that are accessed not by sequential index numbers, but by keys which can have any sort of value. Such an array is sometimes called a *hash* or *dictionary*; in the ruby world, we prefer the term hash. A hash can be constructed by quoting pairs of items within curly braces (`{}`). You use a key to find something in a hash, much as you use an index to find something in an array.

```

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

2.4.1.7 On Control Structures

The case Statement

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

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

Testing For A Range Of Values

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

The Relationship Operator

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

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

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

The while Statement

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

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

Negated Conditions

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

Interrupting A Loop

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

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

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

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

The for Statement

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

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

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

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

for Equivalent To each

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

```
# If you're used to C or Java, you might prefer this.
```

```

for element in collection
  ...
end

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

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

2.4.1.8 Ruby User's Guide On Iterators

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

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

Ruby Allows Us To Define Iterators

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

Iterators In String

Ruby's String type has some useful iterators:

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

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

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

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

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

```

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

Control Structures `retry` And `redo`

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

`yield` In Iterators

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

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

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

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

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

Summary On Iterarors

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

2.4.1.9 On Object-Oriented Thinking

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

Traditionally, a programming problem is attacked by coming up with some kinds of *data representations*, and *procedures* that operate on that data. Under this model, data is inert,

passive, and helpless; it sits at the complete mercy of a large procedural body, which is active, logical, and all-powerful.

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

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

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

What is described above as a “machine” may be very simple or complex on the inside; we can’t tell from the outside, and we don’t allow ourselves to open the machine up (except when we are absolutely sure something is wrong with its design), so we are required to do things like flip the switches and read the dials to interact with the data. Once the machine is built, we don’t want to have to think about how it operates.

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

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

2.4.1.10 On Methods

What Is A Method?

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

Invoking Methods Of An Object

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

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

Polymorphism

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

Errors Are Raised

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

Arguments To A Method

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

The Special Variable `self`

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

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

2.4.1.11 On Classes

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

Making An Object From A Class

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

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

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

Make A New Instance From A Class Definition

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

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

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

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

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

2.4.1.12 On Inheritance

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

We can express this concept in ruby:

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

```
| end  
nil
```

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

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

Differential Programming

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

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

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

2.4.1.13 On Redefinition of Methods

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

```
ruby> class Human  
|   def identify  
|     puts "I'm a person."  
|   end  
end
```

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

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

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

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

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

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

2.4.1.14 On Access Control

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

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

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

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

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

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

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

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

Rationale For Private Methods: Encapsulation

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

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

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

2.4.1.15 On Singleton Methods

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

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

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

2.4.1.16 On Modules

Modules in ruby are similar to classes, except:

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

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

Module As Collection

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

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

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

```
ruby> include Math
Object
ruby> sqrt(2)
1.41421
ruby> PI
3.14159
```

Module As Mixin

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

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

Mixin As Properties

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

Modules Instead Of Multiple Inheritance

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

2.4.1.17 On Procedure Objects (Procs)

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

A new procedure object is formed using `proc`:

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

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

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

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

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

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

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

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

Anonymous Procedure Objects

A final note before we move on to other topics: it's not strictly necessary to give a procedure object a name before binding it to a signal. An equivalent anonymous procedure object would look like `'trap "SIGINT", proc{ puts "^C was pressed." }'`, or more compactly

still, `'trap "SIGINT", 'puts "^C was pressed."'`. This abbreviated form provides some convenience and readability when you write small anonymous procedures.

2.4.1.18 On Variables

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

No Variable Declarations

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

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

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

Table 2.1: List of Variable Identifiers

Pseudo-Variables

The only exceptions to the above are ruby's pseudo-variables: `self`, which always refers to the currently executing object, and `nil`, which is the meaningless value assigned to uninitialized variables. Both are named as if they are local variables, but `self` is a global variable maintained by the interpreter, and `nil` is really a constant. As these are the only two exceptions, they don't confuse things too much.

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

2.4.1.19 On Global Variables

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

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

Global Variables Can Be Traced

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

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

```

ruby> $x = 5
$x is now 5
5

```

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

List Of Major System Variables

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

Table 2.2: List of Major System Variables

2.4.1.20 On Instance Variables

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

Instance Variables Are Not Declared

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

2.4.1.21 On Local Variables

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

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

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

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

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

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

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

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

Proc Objects Are Closures

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

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

```

ruby> writer.call(2)
2
ruby> reader.call
2

```

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

```

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

```

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

2.4.1.22 On Class Constants

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

Class Constants Accessible Outside Class

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

```

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

```

```
101 102 103
nil
```

Constants can also be defined in modules.

```
ruby> module ConstModule
  |   C1=101
  |   C2=102
  |   C3=103
  |   def showConstants
  |     puts "#{C1} #{C2} #{C3}"
  |   end
  | end
nil
ruby> C1
ERR: (eval):1: uninitialized constant C1
ruby> include ConstModule
Object
ruby> C1
101
ruby> showConstants
101 102 103
nil
ruby> C1=99 # not really a good idea
99
ruby> C1
99
ruby> ConstModule::C1
101
ruby> ConstModule::C1=99 # .. this was not allowed in earlier versions
(eval):1: warning: already initialized constant C1
99
ruby> ConstModule::C1 # "enough rope to shoot yourself in the foot"
99
```

2.4.1.23 On Exception Processing and rescue

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

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

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

begin And rescue Blocks

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

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

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

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

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

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

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

raising Exceptions

Every ruby library raises an exception if any error occurs, and you can raise exceptions explicitly in your code too. To raise an exception, use **raise**. It takes one argument, which

should be a string that describes the exception. The argument is optional but should not be omitted. It can be accessed later via the special global variable `$!`.

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

2.4.1.24 On Exception Processing And `ensure`

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

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

```
file = open("/tmp/some_file", "w")
begin
  # ... write to the file ...
rescue
  # ... handle the exceptions ...
ensure
  file.close # ... and this always happens.
end
```

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

2.4.1.25 On Accessors

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

Accessors: Writers And Readers

```
ruby> class Fruit
      |   def set_kind(k) # a writer
      |     @kind = k
      |   end
      |   def get_kind   # a reader
```

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

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

```
ruby> class Fruit
|   def kind=(k)
|     @kind = k
|   end
|   def kind
|     @kind
|   end
| end
nil
ruby> f2 = Fruit.new
#<Fruit:0xfd7e7c8c>
ruby> f2.kind = "banana"
"banana"
ruby> f2.kind
"banana"
```

The inspect Method

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

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


to_s And p Methods

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

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

Making Accessors

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

Shortcut	Effect
<code>attr_reader :v</code>	<code>def v; ; end</code>
<code>attr_writer :v</code>	<code>def v=(value); @v=value; end</code>
<code>attr_accessor :v</code>	<code>attr_reader :v; attr_writer :v</code>
<code>attr_accessor :v, :w</code>	<code>attr_accessor :v; attr_accessor :w</code>

Table 2.3: List of Accessor Shortcuts

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

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

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

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

```
ruby> f2
"a rotting banana"
```

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

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

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

2.4.1.26 On Object Initialization

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

Default Argument Values in `initialize`

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

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

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

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

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

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

Object Reflection, Variable-Length Argument Lists, Method Overloading

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

2.4.1.27 On Nuts And Bolts

Statement Delimiters

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

Comments

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

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

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

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

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

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

Organizing Your Code

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

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

```
puts successor(3)

def successor(x)
  x + 1
end
```

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

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

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

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

def scale(x)
  x * 5.0 / 9.0
end

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

So while this may seem less convenient than what you may be used to in Perl or Java, it is less restrictive than trying to write C without prototypes (which would require you to always maintain a partial ordering of what references what). Putting top-level code at the

bottom of a source file always works. And even this is less of an annoyance than it might at first seem. A sensible and painless way to enforce the behavior you want is to define a `main` function at the top of the file, and call it from the bottom.

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

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

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

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

load And require

It also helps that ruby provides tools for breaking complicated programs into readable, reusable, logically related chunks. We have already seen the use of `include` for accessing modules (see [Section 2.4.1.16 “On Modules”](#), page 58). You will also find the `load` and `require` facilities useful.

<code>load</code>	works as if the file it refers to were copied and pasted in (something like the <code>#include</code> preprocessor directive in C).
<code>require</code>	is somewhat more sophisticated, causing code to be loaded at most once and only when needed.

2.4.2 Programming Ruby

The Programmatic Programmer’s Guide

[Programming Ruby](#)

What This Book Is

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

What Ruby Is

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

You end up with Ruby.

Ruby is OO

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

Ruby is Perl and Python

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

Principle of Least Surprise

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

Ruby is a “Transparent” Language

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

Ruby is a “Scripting” Language

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

Ruby is a General Purpose Programming Language

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

Should I Use Ruby?

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

2.5 Reference Documentation

2.6 Editors and IDEs

2.7 Further Reading

List of Tables

Table 2.1: List of Variable Identifiers 60

Table 2.2: List of Major System Variables..... 61

Table 2.3: List of Accessor Shortcuts..... 68

Appendix A Utility Programs

A.1 eval.rb

eval.rb

{eval.rb} ≡

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

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

module EvalWrapper

  <eval—EvalWrapper—Constants>

  <eval—EvalWrapper—Indentation Deltas>

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

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

  indent=0
  while true    # Top of main loop.

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

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

```
end # module
```

The following table lists called chunk definition points.

Chunk name	First definition point
<code><eval—EvalWrapper—Constants></code>	See “ <code>eval.rb</code> Module Code”, page 77.
<code><eval—EvalWrapper—Indentation Deltas></code>	See “ <code>eval.rb</code> Indentation Deltas Code”, page 77.
<code><eval—Main—Get Line></code>	See “ <code>eval.rb</code> Main Get Line Code”, page 78.
<code><eval—Main—Process Line></code>	See “ <code>eval.rb</code> Main Process Line Code”, page 78.

A.1.1 eval.rb Module Code

```
<eval—EvalWrapper—Constants> ≡
```

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

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

This chunk is called by `{eval.rb}`; see its first definition at “`eval.rb`”, page 76.

A.1.2 eval.rb Indentation Deltas Code

```
<eval—EvalWrapper—Indentation Deltas> ≡
```

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

```
def EvalWrapper.indentation( code )
  case code

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

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

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

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

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

```

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

    end # case
  end # def

```

This chunk is called by {eval.rb}; see its first definition at “eval.rb”, page 76.

A.1.3 eval.rb Main Get Line Code

<eval—Main—Get Line> ≡

```

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

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

else
  print PrMore             # (appending to previous lines)

end

print Ispace * indent,Code
line = gets
print Norm

```

This chunk is called by {eval.rb}; see its first definition at “eval.rb”, page 76.

A.1.4 eval.rb Main Process Line Code

<eval—Main—Process Line> ≡

<eval—Main—Process Line-If Not Line>

<eval—Main—Process Line-Is Line>

This chunk is called by {eval.rb}; see its first definition at “eval.rb”, page 76.

The following table lists called chunk definition points.

Chunk name	First definition point
<i><eval—Main—Process Line-If Not Line></i>	See “eval.rb If Not Line Code”, page 78.
<i><eval—Main—Process Line-Is Line></i>	See “eval.rb If Is Line Code”, page 79.

A.1.4.1 eval.rb If Not Line Code

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

```

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

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

```

This chunk is called by *<eval—Main—Process Line>*; see its first definition at “[eval.rb Main Process Line Code](#)”, page 78.

A.1.4.2 eval.rb If Is Line Code

<eval—Main—Process Line-Is Line> \equiv

```
else

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

  <eval—Main—Process Line-Is Line_Indentation>

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

end # if not line
```

This chunk is called by *<eval—Main—Process Line>*; see its first definition at “[eval.rb Main Process Line Code](#)”, page 78.

The following table lists called chunk definition points.

Chunk name		First definition point
<i><eval—Main—Process Line_Indentation></i>	<i>Line-Is</i>	See “ eval.rb If Is Line Code ”, page 79.
<i><eval—Main—Process Line_ Worth Evaluating?></i>	<i>Line-Is</i>	See “ eval.rb If Is Line Code ”, page 80.

Indentation

<eval—Main—Process Line-Is Line_Indentation> \equiv

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

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

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

end # begin
```

This chunk is called by *<eval—Main—Process Line-Is Line>*; see its first definition at “[eval.rb If Is Line Code](#)”, page 79.

```

### Something Worth Evaluating? ###
<eval—Main-Process Line-Is Line-Worth Evaluating?> ≡
  # Okay, do we have something worth evaulating?

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

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

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

      break if not line

    end # if

```

This chunk is called by *<eval—Main-Process Line-Is Line>*; see its first definition at “[eval.rb If Is Line Code](#)”, page 79.

Appendix B The Makefile

```
{Makefile} ≡
    <Makefile—Variable Definitions>
    <Makefile—Default Rule>
    <Makefile—TWJR Rules>
    <Makefile—Clean Rules>
```

The following table lists called chunk definition points.

Chunk name	First definition point
<Makefile—Clean Rules>	See “Clean Rules”, page 82.
<Makefile—Default Rule>	See “Default Rule”, page 81.
<Makefile—TWJR Rules>	See “TWJR Rules”, page 81.
<Makefile—Variable Definitions>	See “Makefile Variable Definitions”, page 81.

B.1 Makefile Variable Definitions

```
<Makefile—Variable Definitions> ≡
    FILE := Ruby2_5
    SHELL := /bin/bash
```

This chunk is called by {Makefile}; see its first definition at “The Makefile”, page 81.

B.2 Default Rule

The **default** rule is to create a PDF document and all HTML files. This assumes that the TEXI file has been generated and updated by hand first. Therefore, the target TWJR will run both **jrtangle** and **jrweave**, while the target WEAVE or alternatively TEXI will run just **jrweave** on the .twjr file. Thereafter, you can update the .texi file and run the **default**.

```
<Makefile—Default Rule> ≡
    .PHONY : default TWJR TANGLE WEAVE TEXI PDF HTML
    .PHONY : twjr tangle weave texi pdf html
    default : PDF HTML
```

This chunk is called by {Makefile}; see its first definition at “The Makefile”, page 81.

B.3 TWJR Rules

```
<Makefile—TWJR Rules> ≡
    TWJR : twjr
    twjr : tangle weave

    TANGLE : tangle
    tangle : $(FILE).twjr
             jrtangle $(FILE).twjr

    WEAVE : weave
```

```

weave : TEXI
TEXI  : texi
texi   : $(FILE).texi

$(FILE).texi : $(FILE).twjr
    jrweave $(FILE).twjr > $(FILE).texi

PDF : pdf
pdf  : $(FILE).pdf
$(FILE).pdf : $(FILE).texi
    pdftexi2dvi $(FILE).texi
    make distclean

HTML : html
html  : $(FILE)/
$(FILE)/ : $(FILE).texi
    makeinfo --html $(FILE).texi

```

This chunk is called by {Makefile}; see its first definition at [“The Makefile”, page 81](#).

B.4 Clean Rules

<Makefile—Clean Rules> ≡

```

.PHONY : clean distclean veryclean worldclean
clean :
    rm -f *~ \#*\#

distclean : clean
    rm -f *.{aux,log,toc,cp,cps,pg,pgs}

veryclean : clean
    for file in *; do [[ $$file =~ $(FILE)|Makefile ]] && : || rm -vrf $$file ; done;

worldclean : veryclean
    rm -fr $(FILE).{texi,info,pdf} $(FILE)/

```

This chunk is called by {Makefile}; see its first definition at [“The Makefile”, page 81](#).

Appendix C Code Chunk Summaries

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

C.1 Source File Definitions

`{Makefile}`

This chunk is defined in “The Makefile”, page 81.

`{eval.rb}`

This chunk is defined in “eval.rb”, page 76.

`{fact.rb}`

This chunk is defined in “On Simple Examples”, page 39.

`{guess.rb}`

This chunk is defined in “On Puzzle Program”, page 42.

`{regx.rb}`

This chunk is defined in “Regular Expressions”, page 44.

`{ri20min.rb}`

This chunk is defined in “Large Class Definition”, page 21.

C.2 Code Chunk Definitions

<Makefile—Clean Rules>

This chunk is defined in “Clean Rules”, page 82.

<Makefile—Default Rule>

This chunk is defined in “Default Rule”, page 81.

<Makefile—TWJR Rules>

This chunk is defined in “TWJR Rules”, page 81.

<Makefile—Variable Definitions>

This chunk is defined in “Makefile Variable Definitions”, page 81.

<MegaGreeter—Initialize Method>

This chunk is defined in “Large Class Definition”, page 22.

<MegaGreeter—Main Script>

This chunk is defined in “Large Class Definition”, page 23.

<MegaGreeter—say_bye Method>

This chunk is defined in “Large Class Definition”, page 23.

<MegaGreeter—say_hi Method>

This chunk is defined in “Large Class Definition”, page 22.

<eval—EvalWrapper—Constants>

This chunk is defined in “eval.rb Module Code”, page 77.

<eval—EvalWrapper-Indentation Deltas>

This chunk is defined in “[eval.rb Indentation Deltas Code](#)”, page 77.

<eval—Main-Get Line>

This chunk is defined in “[eval.rb Main Get Line Code](#)”, page 78.

<eval—Main-Process Line>

This chunk is defined in “[eval.rb Main Process Line Code](#)”, page 78.

<eval—Main-Process Line-If Not Line>

This chunk is defined in “[eval.rb If Not Line Code](#)”, page 78.

<eval—Main-Process Line-Is Line>

This chunk is defined in “[eval.rb If Is Line Code](#)”, page 79.

<eval—Main-Process Line-Is Line-Indentation>

This chunk is defined in “[eval.rb If Is Line Code](#)”, page 79.

<eval—Main-Process Line-Is Line-Worth Evaluating?>

This chunk is defined in “[eval.rb If Is Line Code](#)”, page 80.

C.3 Code Chunk References

<Makefile—Clean Rules>

This chunk is called by {[Makefile](#)}; see its first definition at “[The Makefile](#)”, page 81.

<Makefile—Default Rule>

This chunk is called by {[Makefile](#)}; see its first definition at “[The Makefile](#)”, page 81.

<Makefile—TWJR Rules>

This chunk is called by {[Makefile](#)}; see its first definition at “[The Makefile](#)”, page 81.

<Makefile—Variable Definitions>

This chunk is called by {[Makefile](#)}; see its first definition at “[The Makefile](#)”, page 81.

<MegaGreeter—Initialize Method>

This chunk is called by {[ri20min.rb](#)}; see its first definition at “[Large Class Definition](#)”, page 21.

<MegaGreeter—Main Script>

This chunk is called by {[ri20min.rb](#)}; see its first definition at “[Large Class Definition](#)”, page 21.

<MegaGreeter—say-bye Method>

This chunk is called by {[ri20min.rb](#)}; see its first definition at “[Large Class Definition](#)”, page 21.

<MegaGreeter—say-hi Method>

This chunk is called by {[ri20min.rb](#)}; see its first definition at “[Large Class Definition](#)”, page 21.

<eval—EvalWrapper—Constants>

This chunk is called by `{eval.rb}`; see its first definition at “`eval.rb`”, page 76.

<eval—EvalWrapper—Indentation Deltas>

This chunk is called by `{eval.rb}`; see its first definition at “`eval.rb`”, page 76.

<eval—Main—Get Line>

This chunk is called by `{eval.rb}`; see its first definition at “`eval.rb`”, page 76.

<eval—Main—Process Line>

This chunk is called by `{eval.rb}`; see its first definition at “`eval.rb`”, page 76.

<eval—Main—Process Line-If Not Line>

This chunk is called by *<eval—Main—Process Line>*; see its first definition at “`eval.rb` Main Process Line Code”, page 78.

<eval—Main—Process Line-Is Line>

This chunk is called by *<eval—Main—Process Line>*; see its first definition at “`eval.rb` Main Process Line Code”, page 78.

<eval—Main—Process Line-Is Line_Indentation>

This chunk is called by *<eval—Main—Process Line-Is Line>*; see its first definition at “`eval.rb` If Is Line Code”, page 79.

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

This chunk is called by *<eval—Main—Process Line-Is Line>*; see its first definition at “`eval.rb` If Is Line Code”, page 79.

Bibliography

Index

"		<MegaGreeter—Initialize Method>, use	21
"#symbol"	12	<MegaGreeter—Main Script>, definition	23
"name".intern	11	<MegaGreeter—Main Script>, use	21
"name".to_sym	11	<MegaGreeter—say_bye Method>, definition	23
		<MegaGreeter—say_bye Method>, use	21
		<MegaGreeter—say_hi Method>, definition	22
		<MegaGreeter—say_hi Method>, use	21
\$			
\$!	65		
+			
++ and --	14	=	
.		== vs equals()	28
.. vs.	13	===	47
:		=~ matching operator	45
:: operator	58	—	
		__FILE__ special variable	23
<			
<eval—EvalWrapper—Constants>, definition	77	'	
<eval—EvalWrapper—Constants>, use	76	"falsey" values	11
<eval—EvalWrapper—Indentation		"truthy" values	11
Deltas>, definition	77		
<eval—EvalWrapper—Indentation Deltas>, use ..	76	\	
<eval—Main—Get Line>, definition	78	\Z	16
<eval—Main—Get Line>, use	76		
<eval—Main—Process Line-If Not		{	
Line>, definition	78	{eval.rb}, definition	76
<eval—Main—Process Line-If Not Line>, use	78	{fact.rb}, definition	39
<eval—Main—Process Line-Is Line>, definition ..	79	{guess.rb}, definition	42
<eval—Main—Process Line-Is Line>, use	78	{Makefile}, definition	81
<eval—Main—Process Line-Is		{regx.rb}, definition	44
Line_Indentation>, definition	79	{ri20min.rb}, definition	21
<eval—Main—Process Line-Is			
Line_Indentation>, use	79		
<eval—Main—Process Line-Is Line_ Worth			
Evaluating?>, definition	80		
<eval—Main—Process Line-Is Line_ Worth			
Evaluating?>, use	79		
<eval—Main—Process Line>, definition	78		
<eval—Main—Process Line>, use	76		
<Makefile—Clean Rules>, definition	82		
<Makefile—Clean Rules>, use	81		
<Makefile—Default Rule>, definition	81		
<Makefile—Default Rule>, use	81		
<Makefile—TWJR Rules>, definition	81		
<Makefile—TWJR Rules>, use	81		
<Makefile—Variable Definitions>, definition	81		
<Makefile—Variable Definitions>, use	81		
<MegaGreeter—Initialize Method>, definition	22		

A

access control	28, 56
access modifier scope	34
access modifiers public , private , protected	34
accessors	66
accessors, using shortcuts	68
active variable	60
Algol and Ruby	40
anonymous procedure objects	59
argument lists, variable length	70
argument values, setting default values	69
ArgumentError , after calling super	15
arguments to a method	52
arithmetic	16
Array	27
array literals in brackets	26
array, converting to and from string	46
array, creating	46
array, referring to elements	46
array, sum elements in	17
arrays	45
arrays are dynamic and mutable	26
arrays, adding	26
arrays, concatenating	46
arrays, repeating	46
associative array	46
attr_accessor :name	20
attr_accessor , methods defined	21
attribute accessors	66
attributes	27, 66

B

binary Ruby extension modules	14
binding of { ... }	9
binding.local_variable_get (:symbol)	12
block	22, 26
block for iterator	27
block object, passed to iterator	8
block, used in an iterator	9
block_given?	10
blocks	37
boolean context	11
braces, none	26
branches page	5
break	48
break statement	44
buffering	43

C

C library, use	16
call method	59
calling method 2 levels up	15
case conventions, enforced	27
cast	27
casting, none	28
chop	45
chop!	45
chruby	4
class constants	63
class constants in modules	64
class constants, access to	63
class definition	19, 53
class definition, repeating	15
class instance variable?	15
class keyword	19, 53
class methods, defining, 2 ways	15
class methods?	15
class variable @@	33
class variables vs class instance variables	15
class variables?	15
class vs module	15
classes	52
classes, modifying	20
classes, open	36
closures, proc objects	62
CLOS	57
collect	32
collection, looping over a range of	
values using for	48
collection, looping over elements using for	48
command line arguments in ARGV	40
comments	70
conditional expression, false values	11
constant naming convention	33
constant, class	63
constructor	26, 28
container types	27
continuations, using	17
control structures	46
control structures retry and redo	50
conventions, naming	33

D

dangerous, destructive methods	15
debugger for Ruby	16
def	18
def keyword	53
default argument values	69
defined? operator	62
destructive method	15
destructive method vs nondestructive method	45
developing Ruby	7
dictionary	46
differential programming	54
DLLs	14

<code>do ... } while</code>	12
<code>do</code> keyword	26
doc tools.....	29, 30
Documentation	3
documentation tool	27
dot notation	52
duck typing.....	23
Dylan	57
dynamicly typed.....	25

E

<code>each</code>	32
<code>each</code> equivalent to <code>for</code>	48
<code>each</code> method of iterator	9
<code>each_byte</code>	49
<code>each_line</code>	49
Eiffel and Ruby	40
empty string	11
<code>end</code> keyword	53
<code>ensure</code>	66
<code>ensure</code> clause.....	14
equivalence vs the same	28
errors in OO	52
<code>eval</code>	12
<code>eval.rb</code>	41
exception handling	14
exception processing	64, 66
expression vs statement	41
extension modules	25

F

factorial in Ruby.....	39
<code>false</code> and <code>nil</code>	11
<code>FalseClass</code>	11
<code>File</code> object, no reference.....	16
file, copy.....	16
file, count lines in	17
file, line number	16
file, process and update contents	16
files, closing.....	16
files, counting words	16
files, reading vs modifying.....	15
files, sorting by modification time	16
<code>find</code>	32
flush standard output	43
<code>for</code> equivalent to <code>each</code>	48
<code>for</code> loop	48
<code>for</code> statement	48
<code>fork</code> vs <code>thread</code>	16
function pointers.....	13
function-like methods, where from?	14
<code>FXRuby</code>	28

G

garbage collector	27
gemsets, manage different using RVM	4
GitHub, ruby repository	7
global variable <code>\$</code>	33
global variable <code>\$!</code>	65
global variable, tracing	60
global variables	60
green threads vs native threads.....	27
<code>gsub</code> method.....	45
<code>gtk+</code>	16
<code>guess.chop!</code>	43
GUI toolkits	28

H

hash	46
<code>Hash</code>	27
heap.....	26

I

identifier with capital letter, method?	15
<code>if</code> modifier	44
immediate values	11
<code>import</code>	28
<code>include</code>	72
include a module, mixin	58
<code>include</code> statement	58
<code>include</code> vs <code>extend</code>	15
inheritance.....	53
initialization of objects.....	69
<code>initialize</code> method.....	22, 69
<code>initialize</code> , constructor	28
<code>inject</code>	32
insert code into a string	18
<code>inspect</code> method	67
installer, third party	3
instance of a class.....	52
instance variabes, as attributes.....	66
instance variable	19
instance variable <code>@</code>	33
instance variables	61
instance variables, accessing.....	14
instance variables, encapsulation	19
<code>instance_methods(false)</code>	15
instantiating a class.....	53
interactively use Ruby	16
interfaces, none, use mixins	28
invoke method	52
invoking original method after redefinition.....	15
<code>irb</code>	17, 30
issue tracker	7
issue tracking	6
iteration	27, 32
iterator	22
iterator as substitute for <code>for</code> loop	48
iterator method.....	27

iterator, block.....	9
iterator, defining.....	49
iterators.....	8
iterators of String class.....	49
iterators, Ruby User's Guide.....	49

J

javadoc.....	27
join.....	46
join method, respond to.....	23

K

Kernel.....	12
key, for hash.....	46
keyword def	53
keyword new	53
keywords class and end	53

L

lambda as a synonym of Proc	9
lambda method.....	37
line number of input file.....	16
load.....	14, 72
local variable scope.....	61
local variables.....	61
loop.....	12
loop interrupts.....	48
loop, for	48
loops using while	47

M

mailing lists.....	7
manage Rubies using chruby	4
Marshal	16
MatchData#begin and MatchData#end	17
matching operator.....	45
member variables, access to.....	27
memory management.....	27
messages to objects.....	51
method definition.....	53
method invocation.....	52
method overloading.....	70
method parameters.....	18
method redefinition.....	54
method, destructive.....	15
method, invoking.....	14, 18
method_missing method.....	37
methods.....	51
methods are virtual.....	27
methods, defining.....	18
mixin.....	58
mixin example.....	15
mixin technique.....	58
mixins.....	27, 28

module function?.....	15
modules.....	58
modules, class constants.....	64
modules, subclassing?.....	15
multiple inheritance.....	27, 58
multiple installations, manage using RVM.....	4
multiple Rubies, command-line tool uru	4
multithreading.....	27

N

negated conditions.....	47
new keyword.....	53
NewtonScript.....	57
next	48
nil and false , similarities and differences.....	11
nil vs null	28
NilClass	11
null vs nil	28

O

object reference is self	27
Object#instance_methods	19
Object#respond_to?	20
object, create from class definition.....	19
object_id methods.....	32
objects, everything including numbers.....	28
objects, strongly and dynamically typed.....	26
objects, strongly typed.....	27
operators?.....	14

P

p method.....	68
parameters, methods.....	18
parentheses, none for condition expressions.....	26
parentheses, optional.....	18, 19
parentheses, optional for method calls.....	26
parentheses, optional in method calls.....	28
Patch Writer's Guide.....	8
patching of Ruby.....	7
polymorphism.....	52
precedence of or	13
precedence, iterators, different results.....	9
predicate method.....	45
predicate method naming convention.....	45
predicate methods.....	11
private vs protected	14
Proc	37
proc	59
Proc as method argument.....	59
Proc object, passed to iterator.....	8
proc , execute.....	59
Proc , invoke.....	59
Proc.new , followed by call	9
procedure objects.....	59
Procs	59

program output, display using `less` 16
 prototype-based languages 57
 pseudo-variables `self` and `nil` 60
`puts` 43

Q

`Qt` 28

R

raise an error 52
`raise` exception 65
 random number seeds 15
 range expression 47
 range operators .. vs 13
`rbenv` 4
`rbenv` version manager 4
`rdoc` 27, 29, 30
`RDoc` 27
 read one line from standard input 43
 reader accessor 66
 redefine a method 54
`redo` 48, 50
 reflection 70
 regexes 28
 regular expression, escaping a backslash 16
 regular expressions 43
 regular expressions at work 45
 relationship operator `===` 47
 releases 5
 repository, Subversion 7
`require` 14, 28, 72
`rescue` 64
`rescue` clause 14
 respond to message, instance variable 22
`respond_to?` method 22
 repository, GitHub 7
`retry` 50, 65
`return` 48
 return multiple values 15
`return` statement unnecessary 40
`ri` 32
 Rubies, switch between 3
 Ruby core 7
 Ruby Core mailing list 7
 Ruby development, tracking 7
 Ruby Tk 28
 Ruby, what it is 38
 ruby-build plugin 4
 Ruby-GNOME2 28
 RVM version manager 3

S

scope of local variable 61
 scope, access modifiers 34
 script code 23
`Self` 57
`self` 52
`self`, meaning 15
`self`, object reference 27
`send` 12
 shared libraries 14
 simple functions? 14
 singleton class? 15
 singleton method 14
 singleton methods 36, 57
`sort` 32
 source, building 5
`split` 46
 standard input object `stdin` 43
 statement delimiters 70
 statement vs expression 41
 static checking, none 28
 statically typed 25
`stdin` 43
`stdin.gets` 43
 string + and * 41
 String or Symbol 33
 Strings 41
 strings, quoting syntax and semantics 41
 strings, sort alphabetically 16
 strongly typed objects 26
`sub` vs `sub!` 16
 subclass 53
 Subversion 7
 Subversion repository 7
`super` 55
`super` gives `ArgumentError` 15
 superclass 53
 sygils in Ruby 29
 Symbol 32
 Symbol object 11
 Symbol or String 33
 symbol, access value of 12
`symbol.to_s` 12
 symbols 32
 symbols as enumeration values 11
 symbols as hash keys 11
 symbols, unique constants 11

T

tabs, expand into spaces 16
 Tcl/Tk, use 16
 templates 27
 ternary operator 17
 Texinfo document formatting language 1
`thread` vs `fork` 16
 threads, native vs green 27
 Tk, won't work 16

<code>to_i</code> method.....	40
<code>to_s</code> method.....	68
track Ruby development	7
<code>trap</code>	16
<code>trap</code> method, and <code>Proc</code>	59
truth values.....	26
type conversions	27
type declarations, none.....	28

U

unit testing lib.....	27
<code>unless</code>	47
<code>until</code>	47
<code>uru</code>	4

V

value, everthing has one.....	32
variable, active.....	60
variable, class <code>@@</code>	33
variable, instance	61
variable, instance <code>@</code>	33
variable, local.....	61
variable-length argument lists	70
variables, description.....	60
variables, global.....	60

variable, global <code>\$</code>	33
version managers.....	3
versions, multiple installations using <code>rbenv</code>	4
versions, switch between using <code>chruby</code>	4
versions,multiple	3
visibility features	27
visibility, changing	14

W

<code>while</code>	42
<code>while</code> statement	47
writer accessor.....	66
<code>WxRuby</code>	28

X

<code>xforms</code>	16
XML vs YAML	28

Y

YAML vs XML.....	28
<code>yield</code>	22, 37, 50
<code>yield</code> control structor, or statement.....	9
<code>yield</code> control structure in iterator.....	9

Program Index

E

`eval.rb` 76

F

`fact.rb` 39

G

`guess.rb` 42

M

`Makefile` 81

R

`regx.rb` 44