Ruby Language QuickRef

General Syntax Rules

Comments start with a pound/sharp (#) character and go to EOL.

Lines between '=begin' and '=end' are skipped by the interpreter.

Ruby programs are sequence of expressions.

Each expression is delimited by semicolons (;) or newlines unless obviously incomplete (e.g. trailing '+').

Backslashes at the end of line does not terminate expression.

Reserved Words

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

Types

Basic types are numbers, strings, ranges, regexen, symbols, arrays, and hashes. Also included are files because they are used so often.

Numbers

```
123 1_234 123.45 1.2e-3
0xffff (hex) 0b01011 (binary) 0377 (octal)
?a      ASCII character
?\C-a   Control-a
?\M-a   Meta-a
?\M-\C-a Meta-Control-a
```

Strings

In all of the %() cases below, you may use any matching characters or any single character for delimiters. %[], %!!, %@@, etc.

```
'no interpolation'
"#{interpolation} and backslashes\n"
%q(no interpolation)
%Q(interpolation and backslashes)
%(interpolation and backslashes)
`echo command interpretation with interpolation and backslashes`
%x(echo command interpretation with interpolation and backslashes)
```

Backslashes

```
\t (tab), \n (newline), \r (carriage return),
\f (form feed), \b (backspace), \a (bell),
\e (escape), \s (whitespace), \nnn (octal),
\xnn (hexadecimal), \cx (control x),
\C-x (control x), \M-x (meta x),
\M-\C-x (meta control x)
```

Here Docs

```
<<identifier    # interpolation
<<"identifier"  # interpolation
<<'identifier'  # no interpolation
<<-identifier   # interpolation, indented end
<<-"identifier" # interpolation, indented end
<<-'identifier' # no interpolation, indented end
```

Symbols

A symbol (:symbol) is an immutable name used for identifiers, variables, and operators.

Ranges

```
1..10
'a'..'z'
(1..10) === 5   -> true
(1..10) === 15  -> false

# prints lines starting at 'start' and
# ending at 'end'
while gets
  print if /start/../end/
end

class RangeThingy
  def <=>(rhs)
    # ...
  end
  def succ
    # ...
  end
end
range = RangeThingy.new(lower_bound) .. RangeThingy.new(upper_bound)
```

Regexen

```
/normal regex/[xim]
%r|alternate form|[xim]
Regex.new(pattern, options)
```

.	any character except newline
[set]	any single character of set
[^set]	any single character NOT of set
*	0 or more previous regular expression
*?	0 or more previous regular expression (non greedy)
+	1 or more previous regular expression
+?	1 or more previous regular expression (non greedy)
?	0 or 1 previous regular expression
|	alternation
( )	grouping regular expressions
^	beginning of a line or string
$	end of a line or string
#{m,n}	at least m but most n previous regular expression
#{m,n}?	at least m but most n previous regular expression (non greedy)
\A	beginning of a string
\b	backspace (0x08, inside [] only)
\B	non-word boundary
\b	word boundary (outside [] only)
\d	digit, same as[0-9]
\D	non-digit
\S	non-whitespace character
\s	whitespace character[\t\n\r\f]
\W	non-word character
\w	word character[0-9A-Za-z_]
\z	end of a string
\Z	end of a string, or before newline at the end
(?#)	comment
(?:)	grouping without backreferences
(?=)	zero-width positive look-ahead assertion (?!)
(?ix-ix)	turns on/off i/x options, localized in group if any.
(?ix-ix:)	turns on/off i/x options, localized in non-capturing group.

Arrays

```
[1, 2, 3]
%w(foo bar baz)     # no interpolation
%W(foo #{bar} baz)  # interpolation
```

Indexes may be negative, and they index backwards (-1 is the last element).

Hashes

```
{ 1 => 2, 2 => 4, 3 => 6 }
{ expr => expr, ... }
```

Files

Common methods include:

```
File.join(p1, p2, ... pN) => "p1/p2/.../pN© platform independent paths
File.new(path, mode_string="r") => file
File.new(path, mode_num [, perm_num]) => file
File.open(filename, mode_string="r") {|f|file block} -> nil
File.open(filename [, mode_num [, perm_num ]]) {|f|file block} -> nil
IO.foreach(path, sepstring=$/) {|l|line block}
IO.readlines(path) => array
```

Mode Strings

r	Read-only, starts at beginning of file (default mode).
r+	Read-write, starts at beginning of file.
w	Write-only, truncates existing file to zero length or creates a new file for writing.
w+	Read-write, truncates existing file to zero length or creates a new file for reading and writing.
a	Write-only, starts at end of file if file exists, otherwise creates a new file for writing.
a+	Read-write, starts at end of file if file exists, otherwise creates a new file for reading and writing.
b	Binary file mode (may appear with any of the key letters listed above). Only **necessary** for DOS/Windows.

Variables and Constants

```
$global_variable
@instance_variable
[OtherClass::]CONSTANT
local_variable
```

Pseudo-variables

self	the receiver of the current method
nil	the sole instance of NilClass (represents false)
true	the sole instance of TrueClass (typical true value)
false	the sole instance of FalseClass (represents false)
__FILE__	the current source file name.
__LINE__	the current line number in the source file.

Pre-defined Variables

$!	The exception information message set by 'raise'.
$@	Array of backtrace of the last exception thrown.
$&	The string matched by the last successful pattern match in this scope.
$`	The string to the left of the last successful match.
$'	The string to the right of the last successful match.
$+	The last bracket matched by the last successful match.
$1	The Nth group of the last successful match. May be > 1.
$~	The information about the last match in the current scope.
$=	The flag for case insensitive, nil by default.
$/	The input record separator, newline by default.
$\	The output record separator for the print and IO#write. Default is nil.
$,	The output field separator for the print and Array#join.
$;	The default separator for String#split.
$.	The current input line number of the last file that was read.
$<	The virtual concatenation file of the files given on command line.
$>	The default output for print, printf. $stdout by default.
$_	The last input line of string by gets or readline.
$0	Contains the name of the script being executed. May be assignable.
$*	Command line arguments given for the script sans args.
$$	The process number of the Ruby running this script.
$?	The status of the last executed child process.
$:	Load path for scripts and binary modules by load or require.
$"	The array contains the module names loaded by require.
$DEBUG	The status of the -d switch.
$FILENAME	Current input file from $<. Same as $<.filename.
$LOAD_PATH	The alias to the $:.
$stderr	The current standard error output.
$stdin	The current standard input.
$stdout	The current standard output.
$VERBOSE	The verbose flag, which is set by the -v switch.
$-0	The alias to $/.

$-a	True if option -a is set. Read-only variable.
$-d	The alias to $DEBUG.
$-F	The alias to $;.
$-i	In in-place-edit mode, this variable holds the extention, otherwise nil.
$-I	The alias to $:.
$-l	True if option -l is set. Read-only variable.
$-p	True if option -p is set. Read-only variable.
$-v	The alias to $VERBOSE.

Pre-defined Global Constants

TRUE	The typical true value.
FALSE	The false itself.
NIL	The nil itself.
STDIN	The standard input. The default value for $stdin.
STDOUT	The standard output. The default value for $stdout.
STDERR	The standard error output. The default value for $stderr.
ENV	The hash contains current environment variables.
ARGF	The alias to the $<.
ARGV	The alias to the $*.
DATA	The file object of the script, pointing just after __END__.
RUBY_VERSION	The ruby version string (VERSION was depricated).
RUBY_RELEASE_DATE	The relase date string.
RUBY_PLATFORM	The platform identifier.

Expressions

Terms

Terms are expressions that may be a basic type (listed above), a shell command, variable reference, constant reference, or method invocation.

Operators and Precedence

```
::
[]
**
- (unary) + (unary) ! ~
*  /  %
+  -
<<  >>
&
|  ^
>  >=  <  <=
<=> == === != =~ !~
&&
||
.. ...
= (+=, -=, ...)
not
and or
```

Control Expressions

```
if bool-expr [then]
  body
elsif bool-expr [then]
  body
else
  body
end

unless bool-expr [then]
  body
else
  body
end

expr if     bool-expr
expr unless bool-expr

case target-expr
   # (comparisons may be regexen)
  when comparison [, comparison]... [then]
    body
  when comparison [, comparison]... [then]
    body
  ...
[else
  body]
end

while bool-expr [do]
 body
end

until bool-expr [do]
 body
end

begin
 body
end while bool-expr

begin
 body
end until bool-expr

for name[, name]... in expr [do]
  body
end

expr.each do | name[, name]... |
  body
end

expr while bool-expr
expr until bool-expr
```

break	terminates loop immediately.
redo	immediately repeats w/o rerunning the condition.
next	starts the next iteration through the loop.
retry	restarts the loop, rerunning the condition.

Invoking a Method

Nearly everything available in a method invocation is optional, consequently the syntax is very difficult to follow. Here are some examples:

```
method

obj.method

Class::method

method(arg1, arg2)

method(arg1, key1 => val1, key2 => val2, aval1, aval2) { block }

method(arg1, *[arg2, arg3]) becomes: method(arg1, arg2, arg3)

call   := [receiver ('::' | '.')] name [params] [block]
params := ( [param]* [, hash] [*arr] [&proc] )
block  := { body } | do body end
```

Defining a Class

Class names begin with capital characters.

```
class Identifier [ < Superclass ]; ... ; end

    # Singleton classes, or idioclasses;
    # add methods to a single instance
    # obj can be self
class << obj; ...; end
```

Defining a Module

Module names begin with capital characters.

```
module Identifier; ...; end
```

Defining a Method

```
def method_name(arg_list); ...; end
def expr.method_name(arg_list); ...; end
```

arg_list := ['('] [varname*] ['*' listname] ['&' blockname] [')']

Arguments may have default values (varname = expr).

Method definitions may not be nested.

method_name may be an operator: <=>, ==, ===, =~, <, <=, > >=, +, -, *, /, %, **, <<, >>, ~, +@, -@, [], []= (the last takes two arguments)

Access Restriction

public	totally accessable.
protected	accessable only by instances of class and direct descendants. Even through hasA relationships. (see below)
private	accessable only by instances of class.

Restriction used without arguments set the default access control. Used with arguments, sets the access of the named methods and constants.

```
class A
  protected
  def protected_method; ...; end
end
class B < A
  public
  def test_protected
    myA = A.new
    myA.protected_method
  end
end
b = B.new.test_protected
```

Accessors

Module provides the following utility methods:

attr_reader <attribute>[, <attribute>]...	Creates a read-only accessor for each <attribute>.
attr_writer <attribute>[, <attribute>]...	Creates a write-only accessor for each <attribute>.
attr <attribute> [, <writable>]	Equivalent to "attr_reader <attribute>; attr_writer <attribute> if <writable>"
attr_accessor <attribute>[, <attribute>]...	Equivalent to "attr <attribute>, true" for each argument.

Aliasing

```
alias <old> <new>
```

Creates a new reference to whatever old referred to. old can be any existing method, operator, global. It may not be a local, instance, constant, or class variable.

Blocks, Closures, and Procs

Blocks/Closures

Blocks must follow a method invocation:

```
invocation do ... end
invocation do || ... end
invocation do |arg_list| ... end
invocation { ... }
invocation { || ... }
invocation { |arg_list| ... }
```

Blocks are full closures, remembering their variable context.

Blocks are invoked via yield and may be passed arguments.

Block arguments may not have default parameters.

Brace form ({/}) has higher precedence and will bind to the last parameter if the invocation is made without parentheses.

do/end form has lower precedence and will bind to the invocation even without parentheses.

Proc Objects

See class Proc for more information. Created via:

```
Kernel#proc (or Kernel#lambda)
Proc#new
&block argument on a method
```

Exceptions

```
begin
  expr
[ rescue [ exception_class [ => var ], ... ]
  expr ]
[ else
  expr ]
[ ensure
  expr ]
end

raise [ exception_class, ] [ message ]
```

The default exception_class for rescue is StandardError, not Exception. Raise without an exception_class raises a RuntimeError. All exception classes must inherit from Exception or one of its children (listed below).

StandardError	LocalJumpError, SystemStackError, ZeroDivisionError, RangeError (FloatDomainError), SecurityError, ThreadError, IOError (EOFError), ArgumentError, IndexError, RuntimeError, TypeError, SystemCallError (Errno::*), RegexpError
SignalException	
Interrupt	
NoMemoryError	
ScriptError	LoadError, NameError, SyntaxError, NotImplementedError
SystemExit	

Catch and Throw

```
catch :label do
  expr
  throw :label
end
```