

# Programming Languages — Ruby

IPA Ruby Standardization WG Draft

August 25, 2010

©Information-technology Promotion Agency, Japan 2010



<b>Contents</b>		Page
1	Scope . . . . .	1
2	Normative references . . . . .	1
3	Conformance . . . . .	1
4	Terms and definitions . . . . .	2
5	Notational conventions . . . . .	4
5.1	General description . . . . .	4
5.2	Syntax . . . . .	4
5.2.1	General description . . . . .	4
5.2.2	Productions . . . . .	4
5.2.3	Syntactic term sequences . . . . .	6
5.2.4	Syntactic terms . . . . .	7
5.2.5	Conceptual names . . . . .	9
5.3	Semantics . . . . .	10
5.4	Attributes of execution contexts . . . . .	11
6	Fundamental concepts . . . . .	11
6.1	Objects . . . . .	11
6.2	Variables . . . . .	12
6.2.1	General description . . . . .	12
6.2.2	Instance variables . . . . .	12
6.3	Methods . . . . .	13
6.4	Blocks . . . . .	13
6.5	Classes, singleton classes, and modules . . . . .	13
6.5.1	General description . . . . .	13
6.5.2	Classes . . . . .	14
6.5.3	Singleton classes . . . . .	14
6.5.4	Inheritance . . . . .	15
6.5.5	Modules . . . . .	16
6.6	Boolean values . . . . .	17
7	Execution contexts . . . . .	17
7.1	General description . . . . .	17
7.2	The initial state . . . . .	18
8	Lexical structure . . . . .	19
8.1	General description . . . . .	19
8.2	Program text . . . . .	19
8.3	Line terminators . . . . .	19
8.4	Whitespace . . . . .	20
8.5	Comments . . . . .	21
8.6	End-of-program markers . . . . .	22
8.7	Tokens . . . . .	22
8.7.1	General description . . . . .	22
8.7.2	Keywords . . . . .	23
8.7.3	Identifiers . . . . .	23
8.7.4	Punctuators . . . . .	24
8.7.5	Operators . . . . .	24
8.7.6	Literals . . . . .	25

8.7.6.1	General description . . . . .	25
8.7.6.2	Numeric literals . . . . .	25
8.7.6.3	String literals . . . . .	28
8.7.6.3.1	General description . . . . .	28
8.7.6.3.2	Single quoted strings . . . . .	28
8.7.6.3.3	Double quoted strings . . . . .	29
8.7.6.3.4	Quoted non-expanded literal strings . . . . .	32
8.7.6.3.5	Quoted expanded literal strings . . . . .	34
8.7.6.3.6	Here documents . . . . .	35
8.7.6.3.7	External command execution . . . . .	37
8.7.6.4	Array literals . . . . .	38
8.7.6.5	Regular expression literals . . . . .	41
8.7.6.6	Symbol literals . . . . .	42
9	Scope of variables . . . . .	43
9.1	General description . . . . .	43
9.2	Scope of local variables . . . . .	44
9.3	Scope of global variables . . . . .	44
10	Program structure . . . . .	45
10.1	Program . . . . .	45
10.2	Compound statement . . . . .	45
11	Expressions . . . . .	46
11.1	General description . . . . .	46
11.2	Logical expressions . . . . .	46
11.2.1	General description . . . . .	46
11.2.2	Keyword logical expressions . . . . .	47
11.2.3	Logical NOT expressions . . . . .	47
11.2.4	Logical AND expressions . . . . .	48
11.2.5	Logical OR expressions . . . . .	49
11.3	Method invocation expressions . . . . .	49
11.3.1	General description . . . . .	49
11.3.2	Method arguments . . . . .	55
11.3.3	Blocks . . . . .	58
11.3.4	The <code>super</code> expression . . . . .	61
11.3.5	The <code>yield</code> expression . . . . .	64
11.4	Operator expressions . . . . .	65
11.4.1	General description . . . . .	65
11.4.2	Assignments . . . . .	65
11.4.2.1	General description . . . . .	65
11.4.2.2	Single assignments . . . . .	66
11.4.2.2.1	General description . . . . .	66
11.4.2.2.2	Single variable assignments . . . . .	66
11.4.2.2.3	Scoped constant assignments . . . . .	68
11.4.2.2.4	Single indexing assignments . . . . .	69
11.4.2.2.5	Single method assignments . . . . .	70
11.4.2.3	Abbreviated assignments . . . . .	71
11.4.2.3.1	General description . . . . .	71
11.4.2.3.2	Abbreviated variable assignments . . . . .	71
11.4.2.3.3	Abbreviated indexing assignments . . . . .	72
11.4.2.3.4	Abbreviated method assignments . . . . .	73
11.4.2.4	Multiple assignments . . . . .	73

11.4.2.5	Assignments with <code>rescue</code> modifiers . . . . .	77
11.4.3	Unary operator expressions . . . . .	77
11.4.3.1	General description . . . . .	77
11.4.3.2	The <code>defined?</code> expression . . . . .	78
11.4.4	Binary operator expressions . . . . .	80
11.5	Primary expressions . . . . .	83
11.5.1	General description . . . . .	83
11.5.2	Control structures . . . . .	84
11.5.2.1	General description . . . . .	84
11.5.2.2	Conditional expressions . . . . .	84
11.5.2.2.1	General description . . . . .	84
11.5.2.2.2	The <code>if</code> expression . . . . .	85
11.5.2.2.3	The <code>unless</code> expression . . . . .	86
11.5.2.2.4	The <code>case</code> expression . . . . .	86
11.5.2.2.5	Conditional operator expression . . . . .	88
11.5.2.3	Iteration expressions . . . . .	88
11.5.2.3.1	General description . . . . .	88
11.5.2.3.2	The <code>while</code> expression . . . . .	89
11.5.2.3.3	The <code>until</code> expression . . . . .	90
11.5.2.3.4	The <code>for</code> expression . . . . .	91
11.5.2.4	Jump expressions . . . . .	91
11.5.2.4.1	General description . . . . .	91
11.5.2.4.2	The <code>return</code> expression . . . . .	92
11.5.2.4.3	The <code>break</code> expression . . . . .	93
11.5.2.4.4	The <code>next</code> expression . . . . .	94
11.5.2.4.5	The <code>redo</code> expression . . . . .	95
11.5.2.4.6	The <code>retry</code> expression . . . . .	95
11.5.2.5	The <code>begin</code> expression . . . . .	96
11.5.3	Grouping expression . . . . .	98
11.5.4	Variable references . . . . .	98
11.5.4.1	General description . . . . .	98
11.5.4.2	Constants . . . . .	99
11.5.4.3	Scoped constants . . . . .	100
11.5.4.4	Global variables . . . . .	100
11.5.4.5	Class variables . . . . .	100
11.5.4.6	Instance variables . . . . .	101
11.5.4.7	Local variables or method invocations . . . . .	101
11.5.4.7.1	General description . . . . .	101
11.5.4.7.2	Determination of the type of local variable identifiers . . . . .	101
11.5.4.7.3	Local variables . . . . .	102
11.5.4.7.4	Method invocations . . . . .	102
11.5.4.8	Pseudo variables . . . . .	102
11.5.4.8.1	General description . . . . .	102
11.5.4.8.2	The <code>nil</code> expression . . . . .	103
11.5.4.8.3	The <code>true</code> expression and the <code>false</code> expression . . . . .	103
11.5.4.8.4	The <code>self</code> expression . . . . .	103
11.5.5	Object constructors . . . . .	104
11.5.5.1	Array constructor . . . . .	104
11.5.5.2	Hash constructor . . . . .	104
11.5.5.3	Range constructor . . . . .	105
12	Statements . . . . .	106
12.1	General description . . . . .	106

12.2	The <code>expression</code> statement . . . . .	106
12.3	The <code>if</code> modifier statement . . . . .	107
12.4	The <code>unless</code> modifier statement . . . . .	107
12.5	The <code>while</code> modifier statement . . . . .	107
12.6	The <code>until</code> modifier statement . . . . .	108
12.7	The <code>rescue</code> modifier statement . . . . .	108
13	Classes and modules . . . . .	109
13.1	Modules . . . . .	109
13.1.1	General description . . . . .	109
13.1.2	Module definition . . . . .	109
13.1.3	Module inclusion . . . . .	111
13.2	Classes . . . . .	111
13.2.1	General description . . . . .	111
13.2.2	Class definition . . . . .	111
13.2.3	Inheritance . . . . .	113
13.2.4	Instance creation . . . . .	113
13.3	Methods . . . . .	113
13.3.1	Method definition . . . . .	113
13.3.2	Method parameters . . . . .	115
13.3.3	Method invocation . . . . .	117
13.3.4	Method lookup . . . . .	119
13.3.5	Method visibility . . . . .	120
13.3.5.1	General description . . . . .	120
13.3.5.2	Public methods . . . . .	120
13.3.5.3	Private methods . . . . .	120
13.3.5.4	Protected methods . . . . .	120
13.3.5.5	Visibility change . . . . .	121
13.3.6	The <code>alias</code> statement . . . . .	121
13.3.7	The <code>undef</code> statement . . . . .	122
13.4	Singleton classes . . . . .	123
13.4.1	General description . . . . .	123
13.4.2	Singleton class definition . . . . .	124
13.4.3	Singleton method definition . . . . .	124
14	Exceptions . . . . .	126
14.1	General description . . . . .	126
14.2	Cause of exceptions . . . . .	126
14.3	Exception handling . . . . .	126
15	Built-in classes and modules . . . . .	127
15.1	General description . . . . .	127
15.2	Built-in classes . . . . .	129
15.2.1	Object . . . . .	129
15.2.1.1	General description . . . . .	129
15.2.1.2	Direct superclass . . . . .	129
15.2.1.3	Included modules . . . . .	129
15.2.1.4	Constants . . . . .	129
15.2.1.5	Instance methods . . . . .	130
15.2.1.5.1	Object#initialize . . . . .	130
15.2.2	Module . . . . .	130
15.2.2.1	General description . . . . .	130
15.2.2.2	Direct superclass . . . . .	130

15.2.2.3	Singleton methods . . . . .	130
15.2.2.3.1	Module.constants . . . . .	130
15.2.2.3.2	Module.nesting . . . . .	131
15.2.2.4	Instance methods . . . . .	131
15.2.2.4.1	Module#< . . . . .	131
15.2.2.4.2	Module#<= . . . . .	132
15.2.2.4.3	Module#<=> . . . . .	132
15.2.2.4.4	Module#== . . . . .	132
15.2.2.4.5	Module#=== . . . . .	132
15.2.2.4.6	Module#> . . . . .	133
15.2.2.4.7	Module#>= . . . . .	133
15.2.2.4.8	Module#alias_method . . . . .	133
15.2.2.4.9	Module#ancestors . . . . .	134
15.2.2.4.10	Module#append_features . . . . .	134
15.2.2.4.11	Module#attr . . . . .	135
15.2.2.4.12	Module#attr_accessor . . . . .	135
15.2.2.4.13	Module#attr_reader . . . . .	136
15.2.2.4.14	Module#attr_writer . . . . .	136
15.2.2.4.15	Module#class_eval . . . . .	136
15.2.2.4.16	Module#class_variable_defined? . . . . .	137
15.2.2.4.17	Module#class_variable_get . . . . .	138
15.2.2.4.18	Module#class_variable_set . . . . .	138
15.2.2.4.19	Module#class_variables . . . . .	138
15.2.2.4.20	Module#const_defined? . . . . .	139
15.2.2.4.21	Module#const_get . . . . .	139
15.2.2.4.22	Module#const_missing . . . . .	140
15.2.2.4.23	Module#const_set . . . . .	140
15.2.2.4.24	Module#constants . . . . .	140
15.2.2.4.25	Module#extend_object . . . . .	141
15.2.2.4.26	Module#extended . . . . .	141
15.2.2.4.27	Module#include . . . . .	141
15.2.2.4.28	Module#include? . . . . .	141
15.2.2.4.29	Module#included . . . . .	142
15.2.2.4.30	Module#included_modules . . . . .	142
15.2.2.4.31	Module#initialize . . . . .	142
15.2.2.4.32	Module#initialize_copy . . . . .	143
15.2.2.4.33	Module#instance_methods . . . . .	144
15.2.2.4.34	Module#method_defined? . . . . .	144
15.2.2.4.35	Module#module_eval . . . . .	145
15.2.2.4.36	Module#private . . . . .	145
15.2.2.4.37	Module#protected . . . . .	145
15.2.2.4.38	Module#public . . . . .	145
15.2.2.4.39	Module#remove_class_variable . . . . .	146
15.2.2.4.40	Module#remove_const . . . . .	147
15.2.2.4.41	Module#remove_method . . . . .	147
15.2.2.4.42	Module#undef_method . . . . .	148
15.2.3	Class . . . . .	148
15.2.3.1	General description . . . . .	148
15.2.3.2	Direct superclass . . . . .	148
15.2.3.3	Instance methods . . . . .	148
15.2.3.3.1	Class#initialize . . . . .	148
15.2.3.3.2	Class#initialize_copy . . . . .	149

15.2.3.3.3	Class#new	149
15.2.3.3.4	Class#superclass	150
15.2.4	NilClass	150
15.2.4.1	General description	150
15.2.4.2	Direct superclass	150
15.2.4.3	Instance methods	150
15.2.4.3.1	NilClass#&	150
15.2.4.3.2	NilClass#^	151
15.2.4.3.3	NilClass#	151
15.2.4.3.4	NilClass#nil?	151
15.2.4.3.5	NilClass#to_s	151
15.2.5	TrueClass	151
15.2.5.1	General description	151
15.2.5.2	Direct superclass	152
15.2.5.3	Instance methods	152
15.2.5.3.1	TrueClass#&	152
15.2.5.3.2	TrueClass#^	152
15.2.5.3.3	TrueClass#to_s	152
15.2.5.3.4	TrueClass#	152
15.2.6	FalseClass	153
15.2.6.1	General description	153
15.2.6.2	Direct superclass	153
15.2.6.3	Instance methods	153
15.2.6.3.1	FalseClass#&	153
15.2.6.3.2	FalseClass#^	153
15.2.6.3.3	FalseClass#to_s	153
15.2.6.3.4	FalseClass#	154
15.2.7	Numeric	154
15.2.7.1	General description	154
15.2.7.2	Direct superclass	154
15.2.7.3	Included modules	154
15.2.7.4	Instance methods	154
15.2.7.4.1	Numeric#+@	154
15.2.7.4.2	Numeric#-@	155
15.2.7.4.3	Numeric#abs	155
15.2.7.4.4	Numeric#coerce	155
15.2.8	Integer	156
15.2.8.1	General description	156
15.2.8.2	Direct superclass	157
15.2.8.3	Instance methods	157
15.2.8.3.1	Integer#+	157
15.2.8.3.2	Integer#-	157
15.2.8.3.3	Integer#*	158
15.2.8.3.4	Integer#/	158
15.2.8.3.5	Integer#%	159
15.2.8.3.6	Integer#<=>	160
15.2.8.3.7	Integer#==	160
15.2.8.3.8	Integer#~	161
15.2.8.3.9	Integer#&	161
15.2.8.3.10	Integer#	161
15.2.8.3.11	Integer#^	162
15.2.8.3.12	Integer#<<	162

15.2.8.3.13	Integer#>>	162
15.2.8.3.14	Integer#ceil	162
15.2.8.3.15	Integer#downto	163
15.2.8.3.16	Integer#eql?	163
15.2.8.3.17	Integer#floor	163
15.2.8.3.18	Integer#hash	164
15.2.8.3.19	Integer#next	164
15.2.8.3.20	Integer#round	164
15.2.8.3.21	Integer#succ	164
15.2.8.3.22	Integer#times	164
15.2.8.3.23	Integer#to_f	165
15.2.8.3.24	Integer#to_i	165
15.2.8.3.25	Integer#to_s	165
15.2.8.3.26	Integer#truncate	166
15.2.8.3.27	Integer#upto	166
15.2.9	Float	166
15.2.9.1	General description	166
15.2.9.2	Direct superclass	167
15.2.9.3	Instance methods	167
15.2.9.3.1	Float#+	167
15.2.9.3.2	Float#-	167
15.2.9.3.3	Float#*	168
15.2.9.3.4	Float#/	168
15.2.9.3.5	Float#%	169
15.2.9.3.6	Float#<=>	170
15.2.9.3.7	Float#==	171
15.2.9.3.8	Float#ceil	171
15.2.9.3.9	Float#finite?	171
15.2.9.3.10	Float#floor	172
15.2.9.3.11	Float#infinite?	172
15.2.9.3.12	Float#round	172
15.2.9.3.13	Float#to_f	172
15.2.9.3.14	Float#to_i	173
15.2.9.3.15	Float#truncate	173
15.2.10	String	173
15.2.10.1	General description	173
15.2.10.2	Direct superclass	173
15.2.10.3	Included modules	173
15.2.10.4	Upper-case and lower-case characters	174
15.2.10.5	Instance methods	174
15.2.10.5.1	String#*	174
15.2.10.5.2	String#+	175
15.2.10.5.3	String#<=>	175
15.2.10.5.4	String#==	176
15.2.10.5.5	String#=~	176
15.2.10.5.6	String#[]	176
15.2.10.5.7	String#capitalize	178
15.2.10.5.8	String#capitalize!	178
15.2.10.5.9	String#chomp	178
15.2.10.5.10	String#chomp!	179
15.2.10.5.11	String#chop	179
15.2.10.5.12	String#chop!	179

15.2.10.5.13	String#downcase	180
15.2.10.5.14	String#downcase!	180
15.2.10.5.15	String#each_line	180
15.2.10.5.16	String#empty?	181
15.2.10.5.17	String#eql?	181
15.2.10.5.18	String#gsub	181
15.2.10.5.19	String#gsub!	183
15.2.10.5.20	String#hash	183
15.2.10.5.21	String#include?	183
15.2.10.5.22	String#index	184
15.2.10.5.23	String#initialize	184
15.2.10.5.24	String#initialize_copy	185
15.2.10.5.25	String#intern	185
15.2.10.5.26	String#length	185
15.2.10.5.27	String#match	185
15.2.10.5.28	String#replace	186
15.2.10.5.29	String#reverse	186
15.2.10.5.30	String#reverse!	186
15.2.10.5.31	String#rindex	186
15.2.10.5.32	String#scan	187
15.2.10.5.33	String#size	188
15.2.10.5.34	String#slice	188
15.2.10.5.35	String#split	188
15.2.10.5.36	String#sub	190
15.2.10.5.37	String#sub!	190
15.2.10.5.38	String#to_i	191
15.2.10.5.39	String#to_f	192
15.2.10.5.40	String#to_s	192
15.2.10.5.41	String#to_sym	192
15.2.10.5.42	String#upcase	192
15.2.10.5.43	String#upcase!	193
15.2.11	Symbol	193
15.2.11.1	General description	193
15.2.11.2	Direct superclass	193
15.2.11.3	Instance methods	193
15.2.11.3.1	Symbol#===	193
15.2.11.3.2	Symbol#id2name	194
15.2.11.3.3	Symbol#to_s	194
15.2.11.3.4	Symbol#to_sym	194
15.2.12	Array	194
15.2.12.1	General description	194
15.2.12.2	Direct superclass	195
15.2.12.3	Included modules	195
15.2.12.4	Singleton methods	195
15.2.12.4.1	Array.[]	195
15.2.12.5	Instance methods	195
15.2.12.5.1	Array#*	195
15.2.12.5.2	Array#+	196
15.2.12.5.3	Array#<<	196
15.2.12.5.4	Array#[]	196
15.2.12.5.5	Array#[]=	197
15.2.12.5.6	Array#clear	198

15.2.12.5.7	Array#collect!	198
15.2.12.5.8	Array#concat	198
15.2.12.5.9	Array#delete_at	199
15.2.12.5.10	Array#each	199
15.2.12.5.11	Array#each_index	199
15.2.12.5.12	Array#empty?	200
15.2.12.5.13	Array#first	200
15.2.12.5.14	Array#index	201
15.2.12.5.15	Array#initialize	201
15.2.12.5.16	Array#initialize_copy	202
15.2.12.5.17	Array#join	202
15.2.12.5.18	Array#last	203
15.2.12.5.19	Array#length	203
15.2.12.5.20	Array#map!	204
15.2.12.5.21	Array#pop	204
15.2.12.5.22	Array#push	204
15.2.12.5.23	Array#replace	204
15.2.12.5.24	Array#reverse	204
15.2.12.5.25	Array#reverse!	205
15.2.12.5.26	Array#rindex	205
15.2.12.5.27	Array#shift	205
15.2.12.5.28	Array#size	206
15.2.12.5.29	Array#slice	206
15.2.12.5.30	Array#unshift	206
15.2.13	Hash	206
15.2.13.1	General description	206
15.2.13.2	Direct superclass	207
15.2.13.3	Included modules	207
15.2.13.4	Instance methods	207
15.2.13.4.1	Hash#==	207
15.2.13.4.2	Hash#[]	208
15.2.13.4.3	Hash#[]=	208
15.2.13.4.4	Hash#clear	209
15.2.13.4.5	Hash#default	209
15.2.13.4.6	Hash#default=	209
15.2.13.4.7	Hash#default_proc	210
15.2.13.4.8	Hash#delete	210
15.2.13.4.9	Hash#each	210
15.2.13.4.10	Hash#each_key	211
15.2.13.4.11	Hash#each_value	211
15.2.13.4.12	Hash#empty?	211
15.2.13.4.13	Hash#has_key?	211
15.2.13.4.14	Hash#has_value?	212
15.2.13.4.15	Hash#include?	212
15.2.13.4.16	Hash#initialize	212
15.2.13.4.17	Hash#initialize_copy	213
15.2.13.4.18	Hash#key?	213
15.2.13.4.19	Hash#keys	213
15.2.13.4.20	Hash#length	214
15.2.13.4.21	Hash#member?	214
15.2.13.4.22	Hash#merge	214
15.2.13.4.23	Hash#replace	215

15.2.13.4.24	Hash#shift	215
15.2.13.4.25	Hash#size	215
15.2.13.4.26	Hash#store	216
15.2.13.4.27	Hash#value?	216
15.2.13.4.28	Hash#values	216
15.2.14	Range	216
15.2.14.1	General description	216
15.2.14.2	Direct superclass	216
15.2.14.3	Included modules	217
15.2.14.4	Instance methods	217
15.2.14.4.1	Range#==	217
15.2.14.4.2	Range#===	217
15.2.14.4.3	Range#begin	218
15.2.14.4.4	Range#each	218
15.2.14.4.5	Range#end	219
15.2.14.4.6	Range#exclude_end?	219
15.2.14.4.7	Range#first	219
15.2.14.4.8	Range#include?	219
15.2.14.4.9	Range#initialize	219
15.2.14.4.10	Range#last	220
15.2.14.4.11	Range#member?	220
15.2.15	Regexp	220
15.2.15.1	General description	220
15.2.15.2	Direct superclass	221
15.2.15.3	Constants	221
15.2.15.4	Patterns	221
15.2.15.5	Matching process	225
15.2.15.6	Singleton methods	226
15.2.15.6.1	Regexp.compile	226
15.2.15.6.2	Regexp.escape	226
15.2.15.6.3	Regexp.last_match	227
15.2.15.6.4	Regexp.quote	228
15.2.15.7	Instance methods	228
15.2.15.7.1	Regexp#initialize	228
15.2.15.7.2	Regexp#initialize_copy	229
15.2.15.7.3	Regexp#==	229
15.2.15.7.4	Regexp#===	230
15.2.15.7.5	Regexp#=~	230
15.2.15.7.6	Regexp#casefold?	230
15.2.15.7.7	Regexp#match	231
15.2.15.7.8	Regexp#source	231
15.2.16	MatchData	231
15.2.16.1	General description	231
15.2.16.2	Direct superclass	232
15.2.16.3	Instance methods	232
15.2.16.3.1	MatchData#[]	232
15.2.16.3.2	MatchData#begin	232
15.2.16.3.3	MatchData#captures	232
15.2.16.3.4	MatchData#end	233
15.2.16.3.5	MatchData#initialize_copy	233
15.2.16.3.6	MatchData#length	234
15.2.16.3.7	MatchData#offset	234

15.2.16.3.8	MatchData#post_match	234
15.2.16.3.9	MatchData#pre_match	234
15.2.16.3.10	MatchData#size	235
15.2.16.3.11	MatchData#string	235
15.2.16.3.12	MatchData#to_a	235
15.2.16.3.13	MatchData#to_s	235
15.2.17	Proc	236
15.2.17.1	General description	236
15.2.17.2	Direct superclass	236
15.2.17.3	Singleton methods	236
15.2.17.3.1	Proc.new	236
15.2.17.4	Instance methods	236
15.2.17.4.1	Proc#[]	236
15.2.17.4.2	Proc#arity	236
15.2.17.4.3	Proc#call	237
15.2.17.4.4	Proc#clone	238
15.2.17.4.5	Proc#dup	238
15.2.18	Struct	239
15.2.18.1	General description	239
15.2.18.2	Direct superclass	239
15.2.18.3	Singleton methods	239
15.2.18.3.1	Struct.new	239
15.2.18.4	Instance methods	241
15.2.18.4.1	Struct#==	241
15.2.18.4.2	Struct#[]	241
15.2.18.4.3	Struct#[]=	242
15.2.18.4.4	Struct#each	243
15.2.18.4.5	Struct#each_pair	243
15.2.18.4.6	Struct#members	243
15.2.18.4.7	Struct#select	243
15.2.18.4.8	Struct#initialize	244
15.2.18.4.9	Struct#initialize_copy	244
15.2.19	Time	245
15.2.19.1	General description	245
15.2.19.2	Direct superclass	245
15.2.19.3	Time computation	245
15.2.19.3.1	Day	245
15.2.19.3.2	Year	246
15.2.19.3.3	Month	247
15.2.19.3.4	Days of month	247
15.2.19.3.5	Hours, Minutes, and Seconds	248
15.2.19.4	Time zone and Local time	248
15.2.19.5	Daylight saving time	248
15.2.19.6	Singleton methods	248
15.2.19.6.1	Time.at	248
15.2.19.6.2	Time.gm	249
15.2.19.6.3	Time.local	251
15.2.19.6.4	Time.mktime	251
15.2.19.6.5	Time.now	251
15.2.19.6.6	Time.utc	252
15.2.19.7	Instance methods	252
15.2.19.7.1	Time#+	252

15.2.19.7.2	Time#-	252
15.2.19.7.3	Time#<=>	253
15.2.19.7.4	Time#asctime	253
15.2.19.7.5	Time#ctime	254
15.2.19.7.6	Time#day	254
15.2.19.7.7	Time#dst?	255
15.2.19.7.8	Time#getgm	255
15.2.19.7.9	Time#getlocal	255
15.2.19.7.10	Time#getutc	255
15.2.19.7.11	Time#gmt?	255
15.2.19.7.12	Time#gmt_offset	256
15.2.19.7.13	Time#gmtime	256
15.2.19.7.14	Time#gmtoff	256
15.2.19.7.15	Time#hour	256
15.2.19.7.16	Time#initialize	256
15.2.19.7.17	Time#initialize_copy	257
15.2.19.7.18	Time#localtime	257
15.2.19.7.19	Time#mday	257
15.2.19.7.20	Time#min	258
15.2.19.7.21	Time#mon	258
15.2.19.7.22	Time#month	258
15.2.19.7.23	Time#sec	258
15.2.19.7.24	Time#to_f	259
15.2.19.7.25	Time#to_i	259
15.2.19.7.26	Time#usec	259
15.2.19.7.27	Time#utc	259
15.2.19.7.28	Time#utc?	260
15.2.19.7.29	Time#utc_offset	260
15.2.19.7.30	Time#wday	260
15.2.19.7.31	Time#yday	261
15.2.19.7.32	Time#year	261
15.2.19.7.33	Time#zone	261
15.2.20	IO	261
15.2.20.1	General description	261
15.2.20.2	Direct superclass	262
15.2.20.3	Included modules	262
15.2.20.4	Singleton methods	263
15.2.20.4.1	IO.open	263
15.2.20.5	Instance methods	263
15.2.20.5.1	IO#close	263
15.2.20.5.2	IO#closed?	264
15.2.20.5.3	IO#each	264
15.2.20.5.4	IO#each_byte	264
15.2.20.5.5	IO#each_line	265
15.2.20.5.6	IO#eof?	265
15.2.20.5.7	IO#flush	265
15.2.20.5.8	IO#getc	266
15.2.20.5.9	IO#gets	266
15.2.20.5.10	IO#initialize_copy	266
15.2.20.5.11	IO#print	266
15.2.20.5.12	IO#putc	267
15.2.20.5.13	IO#puts	267

15.2.20.5.14	IO#read	268
15.2.20.5.15	IO#readchar	269
15.2.20.5.16	IO#readline	269
15.2.20.5.17	IO#readlines	269
15.2.20.5.18	IO#sync	270
15.2.20.5.19	IO#sync=	270
15.2.20.5.20	IO#write	270
15.2.21	File	271
15.2.21.1	General description	271
15.2.21.2	Direct superclass	271
15.2.21.3	Singleton methods	271
15.2.21.3.1	File.exist?	271
15.2.21.4	Instance methods	271
15.2.21.4.1	File#initialize	271
15.2.21.4.2	File#path	272
15.2.22	Exception	272
15.2.22.1	General description	272
15.2.22.2	Direct superclass	273
15.2.22.3	Built-in exception classes	273
15.2.22.4	Singleton methods	273
15.2.22.4.1	Exception.exception	273
15.2.22.5	Instance methods	274
15.2.22.5.1	Exception#exception	274
15.2.22.5.2	Exception#message	274
15.2.22.5.3	Exception#to_s	274
15.2.22.5.4	Exception#initialize	275
15.2.23	StandardError	275
15.2.23.1	General description	275
15.2.23.2	Direct superclass	275
15.2.24	ArgumentError	275
15.2.24.1	General description	275
15.2.24.2	Direct superclass	275
15.2.25	LocalJumpError	275
15.2.25.1	Direct superclass	275
15.2.25.2	Instance methods	275
15.2.25.2.1	LocalJumpError#exit_value	275
15.2.25.2.2	LocalJumpError#reason	276
15.2.26	RangeError	276
15.2.26.1	General description	276
15.2.26.2	Direct superclass	276
15.2.27	RegexpError	276
15.2.27.1	General description	276
15.2.27.2	Direct superclass	276
15.2.28	RuntimeError	276
15.2.28.1	General description	276
15.2.28.2	Direct superclass	276
15.2.29	TypeError	276
15.2.29.1	General description	276
15.2.29.2	Direct superclass	277
15.2.30	ZeroDivisionError	277
15.2.30.1	General description	277
15.2.30.2	Direct superclass	277

15.2.31	NameError	277
15.2.31.1	Direct superclass	277
15.2.31.2	Instance methods	277
15.2.31.2.1	NameError#name	277
15.2.31.2.2	NameError#initialize	277
15.2.32	NoMethodError	278
15.2.32.1	Direct superclass	278
15.2.32.2	Instance methods	278
15.2.32.2.1	NoMethodError#args	278
15.2.32.2.2	NoMethodError#initialize	278
15.2.33	IndexError	279
15.2.33.1	General description	279
15.2.33.2	Direct superclass	279
15.2.34	IOError	279
15.2.34.1	General description	279
15.2.34.2	Direct superclass	279
15.2.35	EOFError	279
15.2.35.1	General description	279
15.2.35.2	Direct superclass	279
15.2.36	SystemCallError	279
15.2.36.1	General description	279
15.2.36.2	Direct superclass	279
15.2.37	ScriptError	279
15.2.37.1	General description	279
15.2.37.2	Direct superclass	279
15.2.38	SyntaxError	280
15.2.38.1	General description	280
15.2.38.2	Direct superclass	280
15.2.39	LoadError	280
15.2.39.1	General description	280
15.2.39.2	Direct superclass	280
15.3	Built-in modules	280
15.3.1	Kernel	280
15.3.1.1	General description	280
15.3.1.2	Singleton methods	280
15.3.1.2.1	Kernel.`	280
15.3.1.2.2	Kernel.block_given?	281
15.3.1.2.3	Kernel.eval	281
15.3.1.2.4	Kernel.global_variables	281
15.3.1.2.5	Kernel.iterator?	282
15.3.1.2.6	Kernel.lambda	282
15.3.1.2.7	Kernel.local_variables	283
15.3.1.2.8	Kernel.loop	283
15.3.1.2.9	Kernel.p	283
15.3.1.2.10	Kernel.print	284
15.3.1.2.11	Kernel.puts	284
15.3.1.2.12	Kernel.raise	284
15.3.1.2.13	Kernel.require	285
15.3.1.3	Instance methods	286
15.3.1.3.1	Kernel#==	286
15.3.1.3.2	Kernel#===	286
15.3.1.3.3	Kernel#_id__	287

15.3.1.3.4	Kernel#_send_ . . . . .	287
15.3.1.3.5	Kernel#` . . . . .	287
15.3.1.3.6	Kernel#block_given? . . . . .	287
15.3.1.3.7	Kernel#class . . . . .	287
15.3.1.3.8	Kernel#clone . . . . .	288
15.3.1.3.9	Kernel#dup . . . . .	288
15.3.1.3.10	Kernel#eql? . . . . .	289
15.3.1.3.11	Kernel#equal? . . . . .	289
15.3.1.3.12	Kernel#eval . . . . .	289
15.3.1.3.13	Kernel#extend . . . . .	289
15.3.1.3.14	Kernel#global_variables . . . . .	290
15.3.1.3.15	Kernel#hash . . . . .	290
15.3.1.3.16	Kernel#initialize_copy . . . . .	290
15.3.1.3.17	Kernel#inspect . . . . .	291
15.3.1.3.18	Kernel#instance_eval . . . . .	291
15.3.1.3.19	Kernel#instance_of? . . . . .	291
15.3.1.3.20	Kernel#instance_variable_defined? . . . . .	292
15.3.1.3.21	Kernel#instance_variable_get . . . . .	292
15.3.1.3.22	Kernel#instance_variable_set . . . . .	292
15.3.1.3.23	Kernel#instance_variables . . . . .	293
15.3.1.3.24	Kernel#is_a? . . . . .	293
15.3.1.3.25	Kernel#iterator? . . . . .	294
15.3.1.3.26	Kernel#kind_of? . . . . .	294
15.3.1.3.27	Kernel#lambda . . . . .	294
15.3.1.3.28	Kernel#local_variables . . . . .	294
15.3.1.3.29	Kernel#loop . . . . .	294
15.3.1.3.30	Kernel#method_missing . . . . .	294
15.3.1.3.31	Kernel#methods . . . . .	295
15.3.1.3.32	Kernel#nil? . . . . .	295
15.3.1.3.33	Kernel#object_id . . . . .	295
15.3.1.3.34	Kernel#p . . . . .	296
15.3.1.3.35	Kernel#print . . . . .	296
15.3.1.3.36	Kernel#private_methods . . . . .	296
15.3.1.3.37	Kernel#protected_methods . . . . .	297
15.3.1.3.38	Kernel#public_methods . . . . .	297
15.3.1.3.39	Kernel#puts . . . . .	297
15.3.1.3.40	Kernel#raise . . . . .	298
15.3.1.3.41	Kernel#remove_instance_variable . . . . .	298
15.3.1.3.42	Kernel#require . . . . .	298
15.3.1.3.43	Kernel#respond_to? . . . . .	299
15.3.1.3.44	Kernel#send . . . . .	299
15.3.1.3.45	Kernel#singleton_methods . . . . .	299
15.3.1.3.46	Kernel#to_s . . . . .	300
15.3.2	Enumerable . . . . .	300
15.3.2.1	General description . . . . .	300
15.3.2.2	Instance methods . . . . .	300
15.3.2.2.1	Enumerable#all? . . . . .	300
15.3.2.2.2	Enumerable#any? . . . . .	301
15.3.2.2.3	Enumerable#collect . . . . .	301
15.3.2.2.4	Enumerable#detect . . . . .	302
15.3.2.2.5	Enumerable#each_with_index . . . . .	302
15.3.2.2.6	Enumerable#entries . . . . .	303

15.3.2.2.7	Enumerable#find	303
15.3.2.2.8	Enumerable#find_all	303
15.3.2.2.9	Enumerable#grep	303
15.3.2.2.10	Enumerable#include?	304
15.3.2.2.11	Enumerable#inject	304
15.3.2.2.12	Enumerable#map	305
15.3.2.2.13	Enumerable#max	305
15.3.2.2.14	Enumerable#min	306
15.3.2.2.15	Enumerable#member?	307
15.3.2.2.16	Enumerable#partition	307
15.3.2.2.17	Enumerable#reject	307
15.3.2.2.18	Enumerable#select	308
15.3.2.2.19	Enumerable#sort	308
15.3.2.2.20	Enumerable#to_a	309
15.3.3	Comparable	309
15.3.3.1	General description	309
15.3.3.2	Instance methods	309
15.3.3.2.1	Comparable#<	309
15.3.3.2.2	Comparable#<=	309
15.3.3.2.3	Comparable#==	310
15.3.3.2.4	Comparable#>	310
15.3.3.2.5	Comparable#>=	310
15.3.3.2.6	Comparable#between?	310

## Introduction

This document specifies the Ruby programming language.

Ruby is an object-oriented scripting language, which has been developed by Yukihiro Matsumoto and his contributors since 1993, and has several implementations distributed as open source software. Ruby has both enough features as an object-oriented language and simplicity as a scripting language, and advanced applications can be implemented with brief code in Ruby. These characteristics of Ruby enables high productivity of program development.

Ruby is thus used for many applications and network systems across the world at the present day, and has multiple implementations. Therefore, a standard specification which underlies compatibility among implementations has been demanded.

The biggest goal of Ruby is developer friendliness, and productivity of application development and intuitive description of program behaviors take precedence over brevity of the language specification itself and ease of implementation. This document is therefore complex as a language specification in order to specify the syntax and semantics of Ruby without ambiguity.



# Information technology — Programming Languages — Ruby

## 1 Scope

This document specifies the syntax and semantics of the computer programming language Ruby and the requirements for conforming Ruby processors, strictly conforming Ruby programs, and conforming Ruby programs.

This document does not specify

- the limit of size or complexity of a program text which is acceptable to a conforming processor,
- the minimal requirements of a data processing system that is capable of supporting a conforming processor,
- the method for activating the execution of programs on a data processing system, and
- the method for reporting syntactic and runtime errors.

NOTE Execution of a Ruby program is to evaluate the *program* (see 10) by a Ruby processor.

## 2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO/IEC 646:1991, *Information technology – ISO 7-bit coded character set for information interchange*.

NOTE Corresponding JIS: JIS X 0201:1997 7-bit and 8-bit coded character sets for information interchange (MOD)

- IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems*.

## 3 Conformance

A strictly conforming Ruby program shall

- 1 • use only those features of the language specified in this document, and
- 2 • not produce output dependent on any unspecified or implementation-defined behavior.

3 A conforming Ruby processor shall

- 4 • accept any strictly conforming programs and behave as specified in this document.

5 A conforming Ruby processor may

- 6 • evaluate a strictly conforming program in a different way from the one described in this  
7 document, if it does not change the behavior of the program; however, if the program  
8 redefines any method of a built-in class or module (see Clause 15), the behavior of the  
9 program may be different from the one described in this document, and

10 NOTE For example, a conforming processor may omit an invocation of a method of a built-in  
11 class or module for optimization purpose, and do the same calculation as the method instead. In  
12 this case, even if a program redefines the method, the behavior of the program might not change  
13 because the redefined method might not actually be invoked.

- 14 • support syntax not described in this document, and accept any programs which use features  
15 not specified in this document.

16 A conforming Ruby program is one that is acceptable to a conforming Ruby processor.

## 17 4 Terms and definitions

18 For the purposes of this document, the following terms and definitions apply. Other terms are  
19 defined where they appear in ***bold slant face*** or on the left side of a syntax rule.

### 20 4.1

#### 21 **block**

22 A procedure which is passed to a method invocation.

### 23 4.2

#### 24 **class**

25 An object which defines the behavior of a set of other objects called its instances.

26 NOTE The behavior is a set of methods which can be invoked on an instance.

### 27 4.3

#### 28 **class variable**

29 A variable whose value is shared by all the instances of a class.

### 30 4.4

#### 31 **constant**

32 A variable which is defined in a class or a module and is accessible both inside and outside the  
33 class or module.

34 NOTE The value of a constant is ordinarily expected to remain unchanged during the execution of a  
35 program, but this document does not force it.

1 **4.5**  
2 **exception**  
3 An object which represents an exceptional event.

4 **4.6**  
5 **global variable**  
6 A variable which is accessible everywhere in a program.

7 **4.7**  
8 **implementation-defined**  
9 Possibly differing between implementations, but defined for every implementation.

10 **4.8**  
11 **instance method**  
12 A method which can be invoked on all the instances of a class.

13 **4.9**  
14 **instance variable**  
15 A variable that exists in a set of variable bindings which every object has.

16 **4.10**  
17 **local variable**  
18 A variable which is accessible only in a certain scope introduced by a program construct such as  
19 a method definition, a block, a class definition, a module definition, a singleton class definition,  
20 or the toplevel of a program.

21 **4.11**  
22 **method**  
23 A procedure which, when invoked on an object, performs a set of computations on the object.

24 **4.12**  
25 **method visibility**  
26 An attribute of a method which determines the conditions under which a method invocation is  
27 allowed.

28 **4.13**  
29 **module**  
30 An object which provides features to be included into a class or another module.

31 **4.14**  
32 **object**  
33 A computational entity which has states and a behavior.

34 NOTE The behavior of an object is a set of methods which can be invoked on the object.

35 **4.15**  
36 **singleton class**  
37 An object which can modify the behavior of its associated object.

38 NOTE A singleton class is ordinarily associated with a single object. However, a conforming processor  
39 may associate a singleton class with multiple objects as described in 13.4.1.

40 **4.16**

1 **singleton method**

2 An instance method of a singleton class.

3 **4.17**

4 **unspecified**

5 Possibly differing between implementations, and not necessarily defined for any particular im-  
6 plementation.

7 **4.18**

8 **variable**

9 A computational entity that refers to an object, which is called the value of the variable.

10 **4.19**

11 **variable binding**

12 An association between a variable and an object which is referred to by the variable.

13 **5 Notational conventions**

14 **5.1 General description**

15 In this clause, the following terms are used:

16 a) sequence of  $A$

17 A “sequence of  $A$ ”, whose length is  $n$ , indicates a sequence whose  $n$  elements  $A_1, A_2, \dots, A_n$   
18 ( $n \geq 0$ ) are of the same kind  $A$ . A sequence whose length is 0 is called an empty sequence.

19 b) sequence of  $A$  separated by  $B$

20 A “sequence of  $A$  separated by  $B$ ”, whose length is  $n + 1$ , indicates a sequence whose  $n + 1$   
21 elements  $A_0, A_1, A_2, \dots, A_n$  ( $n \geq 0$ ) are of the same kind  $A$  and whose adjacent elements  
22 are separated by  $B_1, B_2, \dots, B_n$  of the same kind  $B$  as follows:  $A_0, B_1, A_1, B_2, \dots, B_n,$   
23  $A_n$ .

24 **5.2 Syntax**

25 **5.2.1 General description**

26 In this document, the syntax of the Ruby language is specified by syntactic rules which are  
27 a series of productions (see 5.2.2), and constraints of syntax written in a natural language.  
28 Syntactic rules are given in some subclauses, and are entitled “Syntax”.

29 **5.2.2 Productions**

30 Each production is of the following form, where  $X$  is a nonterminal symbol [see 5.2.4 b)], and  $Y$   
31 is a sequence of syntactic term sequences (see 5.2.3) separated by a vertical line ( $|$ ), and where  
32 whitespace and newlines are used for the sake of readability:

33  $X :: Y$

34 A production defines a set of sequences of characters represented by a nonterminal symbol  $X$   
35 as a union of sets represented by syntactic term sequences in  $Y$ . The production  $X :: Y$  is  
36 therefore called “the production of  $X$ ” or “the  $X$  production.”  $X$  is called the left hand side of

1 the production, and  $Y$  is called the right hand side of the production. The nonterminal symbol  
2  $X$  is said to directly refer to nonterminal symbols which appear in  $Y$ . A relationship that a  
3 nonterminal symbol  $A$  refers to a nonterminal symbol  $B$  is defined recursively as follows:

- 4 • If  $A$  directly refers to  $B$ , then  $A$  refers to  $B$ ;
- 5 • If  $A$  refers to a nonterminal symbol  $C$ , and if  $C$  refers to  $B$ , then  $A$  refers to  $B$ .

6 NOTE 1 A syntactic term represents a set of sequences of characters as described in 5.2.3.

7 In a constraint written in a natural language in a syntactic rule, or in a semantic rule (see 5.3),  
8 “ $X$ ”, where  $X$  is a syntactic term sequence, indicates an element of the set of sequences of  
9 characters represented by the syntactic term sequence  $X$ . Especially in the case that  $X$  is a non-  
10 terminal symbol  $Y$ , “ $Y$ ” indicates an element of the set of sequences of characters represented  
11 by the nonterminal symbol, and “the nonterminal symbol  $Y$ ” indicates the nonterminal symbol  
12 itself. A sequence of characters represented by “ $Y$ ” is also called “of the form  $Y$ .”

13 When a nonterminal symbol  $Y$  directly refers to a nonterminal symbol  $Z$ , “ $Z$  of  $Y$ ” indicates a  
14 part of a sequence of characters represented by  $Y$ , which is represented by such  $Z$ .

15 NOTE 2 For example, a sequence  $x$  of characters represented by  $X$  whose production is “ $X :: Y Z$ ”  
16 consist of a sequence  $y$  of characters represented by  $Y$  and a sequence  $z$  of characters represented by  $Z$ ,  
17 and  $x = yz$ . In this case, “ $Z$  of  $X$ ” indicates  $z$ .

18 “ $Z$  in  $Y$ ” indicates a part of a sequence of characters represented by  $Y$ , which is represented by  
19  $Z$  referred to by the nonterminal symbol  $Y$ .

20 “Each  $Z$  of  $Y$ ” indicates a sequence of characters defined by the following a) to c):

- 21 a) This notation is used when  $Z$  appears in a primary term  $P$  (see 5.2.4), and the right hand  
22 side of the production of  $Y$  contains zero or more repetitions of  $P$  [see 5.2.4 f)] (i.e.,  $P^*$ ).
- 23 b) Let  $Y_n$  ( $n \geq 0$ ) be the right hand side of the production of  $Y$ , where  $P^*$  is replaced with a  
24 sequence of  $P$ s whose length is  $n$ . For any sequence  $y$  of characters represented by  $Y$ , there  
25 exists  $i$  such that a sequence of characters represented by  $Y_i$  is  $y$ .
- 26 c) “Each  $Z$  of  $Y$ ” indicates a part of  $y$  represented by  $Z$  which appears repeatedly in  $Y_i$ .

27 If the number of  $Z$  referred to by  $Y$  in productions in a subclause is only one, “ $Z$ ” is used as a  
28 short form of “ $Z$  of  $Y$ ” or “ $Z$  in  $Y$ .”

29 The nonterminal symbols *input-element* (see 8.1), *program* (see 10.1), and *pattern* (see 15.2.15.4)  
30 are called start symbols.

31 EXAMPLE 1 The following example is the *input-element* production. This production means an *input-*  
32 *element* is any of a *line-terminator*, *whitespace*, *comment*, *end-of-program-marker*, or *token*.

---

33 *input-element* ::  
34     *line-terminator*  
35     | *whitespace*  
36     | *comment*  
37     | *end-of-program-marker*  
38     | *token*

---

1 EXAMPLE 2  $Y$  and  $Z$  are defined as follows:

---

2  $Y ::$   
3  $Z ( \# Z )^*$

4  $Z ::$   
5  $\underline{a} \mid \underline{b} \mid ( Y )$

---

6 In this case, for each following sequence of characters represented by  $Y$ , “each  $Z$  of  $Y$ ” indicates each  
7 underlined part.

8  $\underline{a}$   
9  $\underline{a\#b}$   
10  $\underline{a\#b\#a}$   
11  $\underline{(a\#b)}$   
12  $\underline{a\#(a\#b)\#a}$

### 13 5.2.3 Syntactic term sequences

14 A syntactic term sequence is a sequence of syntactic terms (see 5.2.4). A syntactic term sequence  
15  $S$ , which is a sequence  $T_1 T_2 \dots T_n$  ( $n \geq 1$ ), where  $T_i$  ( $1 \leq i \leq n$ ) is a syntactic term, represents  
16 a set of all sequences of characters of the form  $t_1 t_2 \dots t_n$ , where  $t_i$  is any element of the set of  
17 sequences of characters represented by  $T_i$ . However, if  $T_i$  is a special term, the meaning of  $t_i$  is  
18 defined in 5.2.4 d).

19 *Line-terminators* (see 8.3), *whitespace* (see 8.4), and *comments* (see 8.5) are used to separate  
20 *tokens* (see 8.7), and are ordinarily ignored. *Line-terminators*, *whitespace*, and *comments* are  
21 therefore omitted in the right hand side of productions except in Clause 8 and 15.2.15.4. That  
22 is, in the right hand side of productions, the following syntactic term is omitted before and after  
23 terms.

---

24  $( \textit{line-terminator} \mid \textit{whitespace} \mid \textit{comment} )^*$

---

25 However, a location where a *line-terminator* or *whitespace* shall not occur, or a location where  
26 a *line-terminator* or *whitespace* shall occur is indicated by special terms: a forbidden term [see  
27 5.2.4 d) 2)] or a mandatory term [see 5.2.4 d) 3)], respectively.

28 EXAMPLE The following example represents a sequence of characters: **alias** [a terminal symbol, see  
29 5.2.4 a)], *new-name*, and *aliased-name*, in this order. However, there might be any number of *line-*  
30 *terminators*, *whitespace* characters, and/or *comments* between these elements.

---

31 **alias** *new-name* *aliased-name*

---

1 **5.2.4 Syntactic terms**

2 A syntactic term represents a sequence of characters, or a constraint to a sequence of characters  
3 represented by a syntactic term sequence which includes the syntactic term. A syntactic term  
4 is any of the following a) to h). In particular, syntactic terms a) to c) are called primary terms.

5 NOTE Note that a syntactic term is specified recursively.

6 a) terminal symbol

7 A terminal symbol is shown in **typewriter face**. A terminal symbol represents a set whose  
8 only element is a sequence of characters shown in **typewriter face**.

9 EXAMPLE 1 + represents a sequence of one character “+”. def represents a sequence of three  
10 characters “def”.

11 b) nonterminal symbol

12 A nonterminal symbol is shown in *italic face*. A nonterminal symbol represents a set of  
13 sequences of characters defined by the production of the nonterminal symbol.

14 EXAMPLE 2 A *binary-digit* defined by the following production represents “0” or “1”.

---

15 *binary-digit* ::  
16 0 | 1

---

17 c) grouping term

18 A grouping term is a sequence of syntactic term sequences separated by a vertical line (|)  
19 and enclosed by parentheses [( )]. A grouping term represents a union of sets of sequences  
20 of characters represented by syntactic term sequences in the grouping term.

21 EXAMPLE 3 The following example represents an *alpha-numeric-character* or a *line-terminator*.

---

22 ( *alpha-numeric-character* | *line-terminator* )

---

23 d) special term

24 A special term is a text enclosed by square brackets ([ ]). A special term is any of the  
25 following:

26 1) negative lookahead

27 The notation of a negative lookahead is [lookahead  $\notin$  *S*], where *S* is a sequence of  
28 terminal symbols separated by a comma (,) enclosed by curly brackets ({ }). A negative  
29 lookahead represents a constraint that any sequence of characters in *S* shall not occur  
30 just after the negative lookahead.

31 EXAMPLE 4 The following example means that an *argument-without-parentheses* shall not  
32 begin with “{”:

---

1 *argument-without-parentheses* ::  
2 [lookahead  $\notin$  { { } }] *argument-list*

---

3 2) forbidden term  
4 The notation of a forbidden term is [no *T* here], where *T* is a primary term. A forbidden  
5 term represents a constraint that no *T* shall occur there.

6 EXAMPLE 5 The following example means no *line-terminator* shall occur there.

---

7 [no *line-terminator* here]

---

8 3) mandatory term  
9 The notation of a mandatory term is [*T* here], where *T* is a primary term. A mandatory  
10 term represents a constraint that one or more *T*'s shall occur there.

11 EXAMPLE 6 The following example means one or more *line-terminators* shall occur there.

---

12 [*line-terminator* here]

---

13 4) other special term  
14 The notation of an other special term is [*U*], where *U* is a text which does not match  
15 any of d) 1) to d) 3). This special term represents a set of sequences of characters rep-  
16 resented by *U*, or a constraint represented by *U* to a sequence of characters represented  
17 by a syntactic term sequence which includes this special term.

18 EXAMPLE 7 The following example means that a *source-character* is any character specified  
19 in ISO/IEC 646:1991 IRV:

---

20 *source-character* ::  
21 [ any character in ISO/IEC 646:1991 IRV ]

---

22 EXAMPLE 8 The following example means =begin shall occur at the beginning of a line.

---

23 [ beginning of a line ] =begin

---

24 e) optional term  
25 An optional term is a primary term postfixed with a superscripted question mark (?).

26 An optional term represents a superset of the set represented by the primary term, which  
27 has an empty sequence of characters as the only additional element.

1       EXAMPLE 9   The following example means that the *block* is optional.

---

2               *block*?

---

3 f)   zero or more repetitions

4       A primary term postfixed with a superscripted asterisk (\*) indicates zero or more repetitions  
5       of the primary term. Zero or more repetitions represent a set of sequences of characters  
6       whose elements are all sequences of any zero or more elements of the set represented by the  
7       primary term.

8       EXAMPLE 10   The following example means a sequence of characters which consists of zero or  
9       more *elsif-clauses*.

---

10              *elsif-clause*\*

---

11 g)   one or more repetitions

12       A primary term postfixed with a superscripted plus sign (+) indicates one or more repeti-  
13       tions of the primary term. One or more repetitions represent a set of sequences of characters  
14       whose elements are all sequences of any one or more elements of the set represented by the  
15       primary term.

16       EXAMPLE 11   The following example means a sequence of characters which consists of one or  
17       more *when-clauses*.

---

18              *when-clause*<sup>+</sup>

---

19 h)   exception term

20       An exception term is a sequences of a primary term  $P_1$ , the phrase **but not**, and another  
21       primary term  $P_2$ . An exception term represents a set of sequences of characters whose  
22       elements are all elements of  $P_1$  excluding all elements of  $P_2$ .

23       EXAMPLE 12   The following exmaple represents a *source-character* but not a *single-quoted-string-*  
24       *meta-character*.

---

25              *source-character* **but not** *single-quoted-string-meta-character*

---

## 26   5.2.5   Conceptual names

27   A nonterminal symbol (except start symbols) which is not referred to by any start symbol is  
28   called a conceptual name. In the production of a conceptual name, ::= is used instead of :: to  
29   distinguish conceptual names from other nonterminal symbols.

30   NOTE 1   In this document, some semantically related nonterminal symbols are syntactically away from  
31   each other. Conceptual names are used to define names which organize such nonterminal symbols [e.g.,

1 *assignment* (see 11.4.2]). Conceptual names are also used to define nonterminal symbols used only in  
2 semantic rules [e.g., *binary-operator* (see 11.4.4)].

3 EXAMPLE 1 The following example defines the conceptual name *assignment*, which can be used to  
4 mention either *assignment-expression* or *assignment-statement*.

---

```
5 assignment ::=
6     assignment-expression
7     | assignment-statement
```

---

### 8 5.3 Semantics

9 For syntactic rules, corresponding semantic rules are given in some subclauses, and are entitled  
10 “Semantics”. In this document, the behaviors of programs are specified by processes evaluating  
11 the programs. The evaluation of a program construct, which is a sequence of characters repre-  
12 sented by a nonterminal symbol, usually results in a value, which is called the (resulting) value  
13 of the program construct. Semantic rules specify the ways of evaluating program constructs  
14 specified in corresponding syntactic rules, and the resulting values of the evaluations.

15 The start of evaluation steps of a program construct described in semantic rules is called the  
16 start of the evaluation of the program construct. The time when there is no evaluation step to  
17 be taken for the program construct is called the end of the evaluation of the program construct.  
18 If the evaluation of a program construct has started, and if the evaluation has not ended, the  
19 program construct is said to be under evaluation.

20 If there is no semantic rule corresponding to a nonterminal symbol  $X$ , and if the right hand side  
21 of the production of  $X$  is a sequence of other nonterminal symbols separated by a vertical line  
22 ( $|$ ), the semantic rule of  $X$  is defined by the semantic rules of other nonterminal symbols referred  
23 to by  $X$ .

24 EXAMPLE 1 A *variable* (see 11.5.4) has the following production, and has no description of semantic  
25 rules.

---

```
26 variable ::
27     constant-identifier
28     | global-variable-identifier
29     | class-variable-identifier
30     | instance-variable-identifier
31     | local-variable-identifier
```

---

32 In this case, the semantic rule of *variable* is defined by the semantic rule of *constant-identifier*, *global-*  
33 *variable-identifier*, *class-variable-identifier*, *instance-variable-identifier*, or *local-variable-identifier*.

34 If there is more than one same nonterminal symbol in the right hand side of a production,  
35 the nonterminal symbols have a subscript to distinguish them in semantic rules (e.g., *operator-*  
36 *expression<sub>1</sub>*), if necessary.

37 The semantic rule of a conceptual name describes the semantic rule of program constructs

1 which are elements of the set of sequences of characters represented by the conceptual name. In  
2 semantic rules, “*X*”, where *X* is a conceptual name, indicates a program construct which is an  
3 element of the set of sequences of characters represented by the nonterminal symbol *X*.

4 EXAMPLE 2 *logical-AND-expression* (see 11.2.4) has the following production.

---

5 *logical-AND-expression* ::=  
6     *keyword-AND-expression*  
7     | *operator-AND-expression*

---

8 Since *logical-AND-expression* is a conceptual name, a sequence of characters represented by a *keyword-*  
9 *AND-expression* or *operator-AND-expression* never be recognized as a *logical-AND-expression* under  
10 parsing process of a program text. However, *keyword-AND-expression* and *operator-AND-expression*  
11 have similar semantic rules and they are described as the semantic rule of *logical-AND-expression*. In  
12 semantic rules, “*logical-AND-expression*” indicates a program construct represented by a *keyword-AND-*  
13 *expression* or *operator-AND-expression*.

## 14 5.4 Attributes of execution contexts

15 The names of the attributes of execution contexts (see 7.1) are enclosed in double square brackets  
16 (`[[ ]]`).

17 EXAMPLE `[[self]]` is one of the attributes of execution contexts.

# 18 6 Fundamental concepts

## 19 6.1 Objects

20 An object has states and a behavior. An object has a set of bindings of instance variables (see  
21 6.2.2) as one of its states. Besides the set of bindings of instance variables, an object can have  
22 some attributes as its states, depending on the class of the object. The behavior of an object is  
23 defined by a set of methods (see 6.3) which can be invoked on that object. A method is defined  
24 in a class, a singleton class, or a module (see 6.5).

25 Every value directly manipulated by a program is an object. For example, all of the following  
26 values are objects:

- 27 • A value which is referred to by a variable (see 6.2);
- 28 • A value which is passed to a method as an argument;
- 29 • A value which is returned by a method;
- 30 • A value which is returned as the result of evaluating an *expression* (see Clause 11), a  
31 *statement* (see Clause 12), a *compound-statement* (see 10.2), or a *program* (see 10.1).

32 Other values are not objects, unless explicitly specified as objects.

33 NOTE Primitive values such as integers are also objects. For example, an integer literal (see 8.7.6.2)  
34 evaluates to an object.

## 1 6.2 Variables

### 2 6.2.1 General description

3 A variable is denoted by a name, and refers to an object, which is called the value of the variable.  
4 A variable itself is not an object. While a variable can refer to only one object at a time, an  
5 object can be referred to by more than one variable at a time.

6 A variable is said to be **bound** to an object if the variable refers to the object. This association  
7 of a variable with an object is called a **variable binding**. When a variable with name  $N$  is  
8 bound to an object  $O$ ,  $N$  is called the name of the binding, and  $O$  is called the value of the  
9 binding.

10 There are five kinds of variables:

- 11 • instance variables (see 6.2.2);
- 12 • constants (see 6.5.2);
- 13 • class variables (see 6.5.2);
- 14 • local variables (see 9.2);
- 15 • global variables (see 9.3).

16 Any variable can be bound to any kind of object.

17 EXAMPLE In the following program, first, the local variable `x` refers to an integer, then it refers to a  
18 string, finally it refers to an array.

```
19     x = 123  
20     x = "abc"  
21     x = [1, 2, 3]
```

### 22 6.2.2 Instance variables

23 An object has a set of variable bindings. A variable whose binding is in this set is an instance  
24 variable of that object. This set of bindings of instance variables represents a state of that  
25 object.

26 An instance variable of an object is not directly accessible outside the object. An instance  
27 variable is ordinarily accessed through methods called accessors outside the object. In this  
28 sense, a set of bindings of instance variables is encapsulated in an object.

29 EXAMPLE In the following program, the value of the instance variable `@value` of an instance of the  
30 class `ValueHolder` is initialized by the method `initialize` (see 15.2.3.3.3), and is accessed through the  
31 accessor method `value`, and printed by the method `puts` of the module `Kernel` (see 15.3.1.2.11). Text  
32 after `#` is a comment (see 8.5).

```
33     class ValueHolder  
34         def initialize(value)  
35             @value = value  
36         end  
37
```

```

1     def value
2         return @value
3     end
4 end
5
6     vh = ValueHolder.new(10)    # initialize(10) is invoked.
7     puts vh.value

```

## 8 6.3 Methods

9 A method is a procedure which, when invoked on an object, performs a set of computations on  
10 the object. A method itself is not an object. The behavior of an object is defined by a set of  
11 methods which can be invoked on that object. A method has one or more (when aliased) names  
12 associated with it. An association between a name and a method is called a **method binding**.  
13 When a name *N* is bound to a method *M*, *N* is called the name of the binding, and *M* is called  
14 the value of the binding. A name bound to a method is called the **method name**. A method  
15 can be invoked on an object by specifying one of its names. The object on which the method is  
16 invoked is called the **receiver** of the method invocation.

17 **EXAMPLE** In a method invocation `obj.method(arg1, arg2)`, `obj` is called the receiver, and `method`  
18 is called the method name. See 11.3 for method invocation expressions.

19 Methods are described further in 13.3.

## 20 6.4 Blocks

21 A block is a procedure which is passed to a method invocation. The block passed to a method  
22 invocation is called zero or more times in the method invocation.

23 A block itself is not an object. However, a block can be represented by an object which is an  
24 instance of the class `Proc` (see 15.2.17).

25 **EXAMPLE 1** In the following program, for each element of an array, the block “`{ |i| puts i }`” is  
26 called by the method `each` of the class `Array` (see 15.2.12.5.10).

```

27     a = [1, 2, 3]
28     a.each { |i| puts i }

```

29 **EXAMPLE 2** In the following program, an instance of the class `Proc` which represents the block  
30 “`{ puts "abc" }`” is created, and is called by the method `call` of the class `Proc` (see 15.2.17.4.3).

```

31     x = Proc.new { puts "abc" }
32     x.call

```

33 Blocks are described further in 11.3.3.

## 34 6.5 Classes, singleton classes, and modules

### 35 6.5.1 General description

36 Behaviors of objects are defined by classes, singleton classes, and modules. A class defines  
37 methods shared by objects of the same class. A singleton class is associated to an object, and  
38 can modify the behavior of that object. A module defines, and provides methods to be included  
39 into classes and other modules. Classes, singleton classes, and modules are themselves objects,  
40 which are dynamically created and modified at run-time.

## 1 6.5.2 Classes

2 A class is itself an object, and creates other objects. The created objects are called *direct*  
3 *instances* of the class (see 13.2.4).

4 A class defines a set of methods which, unless overridden (see 13.3.1), can be invoked on all the  
5 instances of the class. These methods are instance methods of the class.

6 A class is itself an object, and created by evaluation of a program construct such as a *class-*  
7 *definition* (see 13.2.2). A class has two sets of variable bindings besides a set of bindings of  
8 instance variables. The one is a set of bindings of constants. The other is a set of bindings of  
9 class variables, which represents the state shared by all the instances of the class.

10 The constants, class variables, singleton methods and instance methods of a class are called the  
11 *features* of the class.

12 EXAMPLE 1 The class `Array` (see 15.2.12) is itself an object, and can be the receiver of a method  
13 invocation. An invocation of the method `new` on the class `Array` creates an object called a direct instance  
14 of the class `Array`.

15 EXAMPLE 2 In the following program, the instance method `push` of the class `Array` (see 15.2.12.5.22)  
16 is invoked on an instance of the class `Array`.

```
17     a = Array.new  
18     a.push(1, 2, 3)    # The value of a is changed to [1, 2, 3].
```

19 EXAMPLE 3 In the following program, the class `X` is defined by a class definition. The class variable  
20 `@@a` is shared by instances of the class `X`.

```
21     class X  
22         @@a = "abc"  
23  
24         def print_a  
25             puts @@a  
26         end  
27  
28         def set_a(value)  
29             @@a = value  
30         end  
31     end  
32     x1 = X.new  
33     x1.print_a      # prints abc  
34     x2 = X.new  
35     x2.set_a("def")  
36     x2.print_a     # prints def  
37     x1.print_a     # prints def
```

38 Classes are described further in 13.2.

## 39 6.5.3 Singleton classes

40 Every object, including classes, can be associated with at most one singleton class. The singleton  
41 class defines methods which can be invoked on that object. Those methods are singleton methods  
42 of the object. If the object is not a class, the singleton methods of the object can be invoked  
43 only on that object. If the object is a class, singleton methods of the class can be invoked only  
44 on that class and its subclasses (see 6.5.4).

1 A singleton class is created, and associated with an object by a singleton class definition (see  
2 13.4.2) or a singleton method definition (see 13.4.3).

3 EXAMPLE 1 In the following program, the singleton class of `x` is created by a singleton class definition.  
4 The method `show` is called a singleton method of `x`, and can be invoked only on `x`.

```
5     x = "abc"
6     y = "def"
7
8     # The definition of the singleton class of x
9     class << x
10      def show
11        puts self    # prints the receiver
12      end
13    end
14
15    x.show    # prints abc
16    y.show    # raises an exception
```

17 EXAMPLE 2 In the following program, the same singleton method `show` as EXAMPLE 1 is defined  
18 by a singleton method definition. The singleton class of `x` is created implicitly by the singleton method  
19 definition.

```
20     x = "abc"
21
22     # The definition of a singleton method of x
23     def x.show
24       puts self    # prints the receiver
25     end
26
27     x.show
```

28 EXAMPLE 3 In the following program, the singleton method `a` of the class `X` is defined by a singleton  
29 method definition.

```
30     class X
31       # The definition of a singleton method of the class X
32       def X.a
33         puts "The method a is invoked."
34       end
35     end
36     X.a
```

37 NOTE Singleton methods of a class is similar to so-called class methods in other object-oriented  
38 languages because they can be invoked on that class.

39 Singleton classes are described further in 13.4.

#### 40 6.5.4 Inheritance

41 A class has at most one single class as its **direct superclass**. If a class `A` has a class `B` as its  
42 direct superclass, `A` is called a **direct subclass** of `B`.

43 All the classes in a program, including built-in classes, form a rooted tree called a **class inher-**  
44 **itance tree**, where the parent of a class is its direct superclass, and the children of a class are  
45 all its direct subclasses. There is only one class which does not have a superclass. It is the root

1 of the tree. All the ancestors of a class in the tree are called **superclasses** of the class. All the  
2 descendants of a class in the tree are called **subclasses** of the class.

3 A class inherits constants, class variables, singleton methods, and instance methods from its  
4 superclasses, if any (see 13.2.3). If an object *C* is a direct instance of a class *D*, *C* is called an  
5 instance of *D* and all its superclasses.

6 **EXAMPLE** The following program defines three classes: the class X, the class Y, and the class Z.

```
7     class X
8     end
9
10    class Y < X
11    end
12
13    class Z < Y
14    end
```

15 The class X is called the direct superclass of the class Y, and the class Y is called a direct subclass of the  
16 class X. The class Y inherits features from the class X. The class X is called a superclass of the class Z, and  
17 the class Z is called a subclass of the class X. The class Z inherits features from the class X and the class  
18 Y. A direct instance of the class Z is called an instance of the class X, the class Y, and the class Z.

### 19 6.5.5 Modules

20 Multiple inheritance of classes is not permitted. That is, a class can have only one direct  
21 superclass. However, features can be appended to a class from multiple modules by using  
22 module inclusions.

23 A module is an object which has the same structure as a class except that it cannot create an  
24 instance of itself and cannot be inherited. As with classes, a module has a set of constants, a  
25 set of class variables, and a set of instance methods. Instance methods, constants, and class  
26 variables defined in a module can be used by other classes, modules, and singleton classes by  
27 including the module into them.

28 While a class can have only one direct superclass, a class, a module, or a singleton class can  
29 include multiple modules. Instance methods defined in a module can be invoked on an instance  
30 of a class which includes the module. A module is created by a module definition (see 13.1.2).

31 **EXAMPLE** The following example is not a strictly conforming Ruby program, because a class cannot  
32 have multiple direct superclasses.

```
33     class Stream
34     end
35
36     class ReadStream < Stream
37     def read(n)
38         # reads n bytes from a stream
39     end
40     end
41
42     class WriteStream < Stream
43     def write(str)
44         # writes str to a stream
45     end
46     end
```

```
1
2   class ReadWriteStream < ReadStream, WriteStream
3   end
```

4 Instead, a class can include multiple modules. The following example uses module inclusion instead of  
5 multiple inheritance.

```
6   class Stream
7   end
8
9   module Readable
10    def read(n); end
11  end
12
13  module Writable
14    def write(str); end
15  end
16
17  class ReadStream < Stream
18    include Readable
19  end
20
21  class WriteStream < Stream
22    include Writable
23  end
24
25  class ReadWriteStream
26    include Readable
27    include Writable
28  end
```

29 Modules are described further in 13.1.

## 30 6.6 Boolean values

31 An object is classified into either a *trueish object* or a *falseish object*.

32 Only **false** and **nil** are falseish objects. **false** is the only instance of the class `FalseClass` (see  
33 15.2.6), to which a *false-expression* evaluates (see 11.5.4.8.3). **nil** is the only instance of the class  
34 `NilClass` (see 15.2.4), to which a *nil-expression* evaluates (see 11.5.4.8.2).

35 Objects other than **false** and **nil** are classified into trueish objects. **true** is the only instance of  
36 the class `TrueClass` (see 15.2.5), to which a *true-expression* evaluates (see 11.5.4.8.3).

## 37 7 Execution contexts

### 38 7.1 General description

39 An *execution context* is a set of attributes which affects evaluation of a program.

40 An execution context is not a part of the Ruby language. It is defined in this document only for  
41 the description of the semantics of a program. A conforming processor shall evaluate a program  
42 producing the same result as if the processor acted within an execution context in the manner  
43 described in this document.

1 An execution context consists of a set of attributes as described below. Each attribute of an  
2 execution context except `[[global-variable-bindings]]` forms a stack. Attributes of an execution  
3 context are changed when a program construct is evaluated.

4 The following are the attributes of an execution context:

5 `[[self]]`: A stack of objects. The object at the top of the stack is called the **current self**,  
6 to which a *self-expression* evaluates (see 11.5.4.8.4).

7 `[[class-module-list]]`: A stack of lists of classes, modules, or singleton classes. The class or  
8 module at the head of the list which is on the top of the stack is called the **current class**  
9 **or module**.

10 `[[default-method-visibility]]`: A stack of visibilities of methods, each of which is one of the  
11 **public**, **private**, and **protected** visibility. The top of the stack is called the **current**  
12 **visibility**.

13 `[[local-variable-bindings]]`: A stack of sets of bindings of local variables. The element at the  
14 top of the stack is called the **current set of local variable bindings**. A set of bindings  
15 is pushed onto the stack on every entry into a local variable scope (see 9.2), and the top  
16 element is removed from the stack on every exit from the scope. The scope with which  
17 an element in the stack is associated is called the **scope of the set of local variable**  
18 **bindings**.

19 `[[invoked-method-name]]`: A stack of names by which methods are invoked.

20 `[[defined-method-name]]`: A stack of names with which the invoked methods are defined.

21 NOTE The top elements of `[[invoked-method-name]]` and `[[defined-method-name]]` are usually the  
22 same. However, they can be different if an invoked method has an alias name.

23 `[[block]]`: A stack of blocks passed to method invocations. An element of the stack may  
24 be block-not-given. **block-not-given** is the special value which indicates that no block is  
25 passed to a method invocation.

26 `[[global-variable-bindings]]`: A set of bindings of global variables.

## 27 7.2 The initial state

28 Immediately prior to execution of a program, the attributes of the execution context is initialized  
29 as follows:

- 30 a) Set `[[global-variable-bindings]]` to a newly created empty set.
- 31 b) Create built-in classes and modules as described in Clause 15.
- 32 c) Create an empty stack for each attribute of the execution context except `[[global-variable-`  
33 `bindings]]`.
- 34 d) Create a direct instance of the class `Object` and push it onto `[[self]]`.
- 35 e) Create a list containing only element, the class `Object`, and push the list onto `[[class-module-`  
36 `list]]`.

- 1 f) Push the private visibility onto `[[default-method-visibility]]`.
- 2 g) Push block-not-given onto `[[block]]`.

## 3 8 Lexical structure

### 4 8.1 General description

#### 5 Syntax

---

```
6 input-element ::  
7     line-terminator  
8     | whitespace  
9     | comment  
10    | end-of-program-marker  
11    | token
```

---

12 The program text of a program is first converted into a sequence of *input-elements*, which are ei-  
13 ther *line-terminators*, *whitespace*, *comments*, *end-of-program-markers*, or *tokens*. When several  
14 prefixes of the input under the converting process have matching productions, the production  
15 that matches the longest prefix is selected.

### 16 8.2 Program text

#### 17 Syntax

---

```
18 source-character ::  
19     [ any character in ISO/IEC 646:1991 IRV ]
```

---

20 A program is represented as a **program text**. A program text is a sequence of *source-characters*.  
21 A *source-character* is a character in ISO/IEC 646:1991 IRV (the International Reference Ver-  
22 sion). The support for any other character sets and encodings is unspecified.

23 Terminal symbols are sequences of those characters in ISO/IEC 646:1991 IRV. Control characters  
24 in ISO/IEC 646:1991 IRV are represented by two digits in hexadecimal notation prefixed by “0x”,  
25 where the first and the second digits respectively represent x and y of the notations of the form  
26 x/y specified in ISO/IEC 646, 5.1.

27 EXAMPLE “0x0a” represents the character LF, whose bit combination specified in ISO/IEC 646 is  
28 0/10.

### 29 8.3 Line terminators

#### 30 Syntax

---

1 *line-terminator* ::  
2     0x0d<sup>?</sup> 0x0a

---

3 Except in Clause 8 and 15.2.15.4, *line-terminators* are omitted from productions as described  
4 in 5.2.3. However, a location where a *line-terminator* shall not occur, or a location where a  
5 *line-terminator* shall occur is indicated by special terms: a forbidden term [see 5.2.4 d) 2)] or a  
6 mandatory term [see 5.2.4 d) 3)], respectively.

7 EXAMPLE *statements* are separated by *separators* (see 10.2). The syntax of the *separators* is as  
8 follows:

---

9 *separator* ::  
10     ;  
11     | [*line-terminator* here]

---

12 The source

```
13     x = 1 + 2  
14     puts x
```

15 is therefore separated into the two *statements* “x = 1 + 2” and “puts x” by a *line-terminator*.

16 The source

```
17     x =  
18     1 + 2
```

19 is parsed as the single *statement* “x = 1 + 2” because “x =” is not a *statement*. However, the source

```
20     x  
21     = 1 + 2
```

22 is not a strictly conforming Ruby program because a *line-terminator* shall not occur before = in a *single-*  
23 *variable-assignment-expression*, and “= 1 + 2” is not a *statement*. The fact that a *line-terminator* shall  
24 not occur before = is indicated in the syntax of the *single-variable-assignment-expression* as follows (see  
25 11.4.2.2.2):

---

26 *single-variable-assignment-expression* ::  
27     *variable* [no *line-terminator* here] = *operator-expression*

---

## 28 8.4 Whitespace

### 29 Syntax

---

1 *whitespace* ::  
2     0x09 | 0x0b | 0x0c | 0x0d | 0x20 | *line-terminator-escape-sequence*

3 *line-terminator-escape-sequence* ::  
4     \*line-terminator*

---

5 Except in Clause 8 and 15.2.15.4, *whitespace* is omitted from productions as described in 5.2.3.  
6 However, a location where *whitespace* shall not occur, or a location where *whitespace* shall occur  
7 is indicated by special terms: a forbidden term [see 5.2.4 d) 2)] or a mandatory term [see 5.2.4  
8 d) 3)] , respectively.

## 9 8.5 Comments

### 10 Syntax

---

11 *comment* ::  
12     *single-line-comment*  
13     | *multi-line-comment*

14 *single-line-comment* ::  
15     # *comment-content*<sup>?</sup>

16 *comment-content* ::  
17     *line-content*

18 *line-content* ::  
19     ( *source-character*<sup>+</sup> ) **but not** ( *source-character*<sup>\*</sup> *line-terminator* *source-character*<sup>\*</sup> )

20 *multi-line-comment* ::  
21     *multi-line-comment-begin-line* *multi-line-comment-line*<sup>?</sup>  
22     *multi-line-comment-end-line*

23 *multi-line-comment-begin-line* ::  
24     [ beginning of a line ] =**begin** *rest-of-begin-end-line*<sup>?</sup> *line-terminator*

25 *multi-line-comment-end-line* ::  
26     [ beginning of a line ] =**end** *rest-of-begin-end-line*<sup>?</sup>  
27     ( *line-terminator* | [ end of a program ] )

28 *rest-of-begin-end-line* ::  
29     *whitespace*<sup>+</sup> *comment-content*

30 *multi-line-comment-line* ::  
31     *comment-line* **but not** *multi-line-comment-end-line*

1 *comment-line* ::  
2 *comment-content line-terminator*

---

3 The notation “[ beginning of a line ]” indicates the beginning of a program or the position  
4 immediately after a *line-terminator*.

5 A *comment* is either a *single-line-comment* or a *multi-line-comment*. Except in Clause 8 and  
6 15.2.15.4, *comments* are omitted from productions as described in 5.2.3.

7 A *single-line-comment* begins with “#” and continues to the end of the line. A *line-terminator*  
8 at the end of the line is not considered to be a part of the comment. A *single-line-comment* can  
9 contain any characters except *line-terminators*.

10 A *multi-line-comment* begins with a line beginning with **=begin**, and continues until and in-  
11 cluding a line that begins with **=end**. Unlike *single-line-comments*, a *line-terminator* of a *multi-*  
12 *line-comment-end-line*, if any, is considered to be part of the comment.

13 NOTE A *line-content* is a sequence of *source-characters*. However, *line-terminators* are not permitted  
14 within a *line-content* as specified in the *line-content* production.

## 15 8.6 End-of-program markers

### 16 Syntax

---

17 *end-of-program-marker* ::  
18 [ beginning of a line ] **\_\_END\_\_** ( *line-terminator* | [ end of a program ] )

---

19 An *end-of-program-marker* indicates the end of a program. Any source characters after an  
20 *end-of-program-marker* are not treated as a program text.

21 NOTE **\_\_END\_\_** is not a *keyword*, and can be a *local-variable-identifier*.

## 22 8.7 Tokens

### 23 8.7.1 General description

#### 24 Syntax

---

25 *token* ::  
26 *keyword*  
27 | *identifier*  
28 | *punctuator*  
29 | *operator*  
30 | *literal*

---

## 1 8.7.2 Keywords

### 2 Syntax

---

3 *keyword* ::  
4     \_\_LINE\_\_ | \_\_ENCODING\_\_ | \_\_FILE\_\_ | BEGIN | END | alias | and | begin  
5     | break | case | class | def | defined? | do | else | elsif | end  
6     | ensure | for | false | if | in | module | next | nil | not | or | redo  
7     | rescue | retry | return | self | super | then | true | undef | unless  
8     | until | when | while | yield

---

9 Keywords are case-sensitive.

10 NOTE \_\_LINE\_\_, \_\_ENCODING\_\_, \_\_FILE\_\_, BEGIN, and END are reserved for future use.

## 11 8.7.3 Identifiers

### 12 Syntax

---

13 *identifier* ::  
14     *local-variable-identifier*  
15     | *global-variable-identifier*  
16     | *class-variable-identifier*  
17     | *instance-variable-identifier*  
18     | *constant-identifier*  
19     | *method-only-identifier*  
20     | *assignment-like-method-identifier*

21 *local-variable-identifier* ::  
22     ( *lowercase-character* | *\_* ) *identifier-character*\*

23 *global-variable-identifier* ::  
24     \$ *identifier-start-character* *identifier-character*\*

25 *class-variable-identifier* ::  
26     @@ *identifier-start-character* *identifier-character*\*

27 *instance-variable-identifier* ::  
28     @ *identifier-start-character* *identifier-character*\*

29 *constant-identifier* ::  
30     *uppercase-character* *identifier-character*\*

31 *method-only-identifier* ::  
32     ( *constant-identifier* | *local-variable-identifier* ) ( ! | ? )

```

1  assignment-like-method-identifier ::
2      ( constant-identifier | local-variable-identifier ) =

3  identifier-character ::
4      lowercase-character
5      | uppercase-character
6      | decimal-digit
7      | -

8  identifier-start-character ::
9      lowercase-character
10     | uppercase-character
11     | -

12 uppercase-character ::
13     A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R
14     | S | T | U | V | W | X | Y | Z

15 lowercase-character ::
16     a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r
17     | s | t | u | v | w | x | y | z

18 decimal-digit ::
19     0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

---

## 20 8.7.4 Punctuators

### 21 Syntax

```

22 punctuator ::
23     [ | ] | ( | ) | { | } | :: | , | ; | .. | ... | ? | : | =>

```

---

## 24 8.7.5 Operators

### 25 Syntax

```

26 operator ::
27     ! | != | !~ | && | ||
28     | operator-method-name
29     | =
30     | assignment-operator

31 operator-method-name ::
32     ^ | & | | | <=> | == | === | =~ | > | >= | < | <= | << | >> | + | -

```

1       | \* | / | % | \*\* | ~ | +@ | -@ | [] | []= | ‘

2     *assignment-operator* ::

3       *assignment-operator-name* =

4     *assignment-operator-name* ::

5       && | || | ^ | & | | | << | >> | + | - | \* | / | % | \*\*

---

## 6   8.7.6   Literals

### 7   8.7.6.1   General description

---

8     *literal* ::

9       *numeric-literal*

10      | *string-literal*

11      | *array-literal*

12      | *regular-expression-literal*

13      | *symbol*

---

### 14 8.7.6.2   Numeric literals

#### 15 Syntax

---

16   *numeric-literal* ::

17      *signed-number*

18      | *unsigned-number*

19   *signed-number* ::

20      ( + | - ) *unsigned-number*

21   *unsigned-number* ::

22      *integer-literal*

23      | *float-literal*

24   *integer-literal* ::

25      *decimal-integer-literal*

26      | *binary-integer-literal*

27      | *octal-integer-literal*

28      | *hexadecimal-integer-literal*

29   *decimal-integer-literal* ::

30      *unprefixed-decimal-integer-literal*

31      | *prefixed-decimal-integer-literal*

1 *unprefixed-decimal-integer-literal* ::  
2     0  
3     | *decimal-digit-except-zero* ( *\_*? *decimal-digit* )\*

4 *prefixed-decimal-integer-literal* ::  
5     0 ( *d* | *D* ) *digit-decimal-part*

6 *digit-decimal-part* ::  
7     *decimal-digit* ( *\_*? *decimal-digit* )\*

8 *binary-integer-literal* ::  
9     0 ( *b* | *B* ) *binary-digit* ( *\_*? *binary-digit* )\*

10 *octal-integer-literal* ::  
11     0 ( *\_* | *o* | *O* )? *octal-digit* ( *\_*? *octal-digit* )\*

12 *hexadecimal-integer-literal* ::  
13     0 ( *x* | *X* ) *hexadecimal-digit* ( *\_*? *hexadecimal-digit* )\*

14 *float-literal* ::  
15     *float-literal-without-exponent*  
16     | *float-literal-with-exponent*

17 *float-literal-without-exponent* ::  
18     *unprefixed-decimal-integer-literal* . *digit-decimal-part*

19 *float-literal-with-exponent* ::  
20     *significand-part* *exponent-part*

21 *significand-part* ::  
22     *float-literal-without-exponent*  
23     | *unprefixed-decimal-integer-literal*

24 *exponent-part* ::  
25     ( *e* | *E* ) ( *+* | *-* )? *digit-decimal-part*

26 *decimal-digit-except-zero* ::  
27     1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

28 *binary-digit* ::  
29     0 | 1

30 *octal-digit* ::  
31     0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

1 *hexadecimal-digit* ::  
2 *decimal-digit* | a | b | c | d | e | f | A | B | C | D | E | F

---

3 If the previous token of a *signed-number* is a *local-variable-identifier*, *constant-identifier*, or  
4 *method-only-identifier*, at least one *whitespace* character or *line-terminator* shall be present be-  
5 tween the *local-variable-identifier*, *constant-identifier*, or *method-only-identifier*, and the *signed-*  
6 *number*.

7 EXAMPLE -123 in the following program is a *signed-number* because there is *whitespace* between x  
8 and -123.

9 x -123

10 In the above program, the method x is invoked with the value of -123 as the argument.

11 However, -123 in the following program is separated into the two tokens - and 123 because there is no  
12 *whitespace* between x and -123.

13 x-123

14 In the above program, the method - is invoked on the value of x with the value of 123 as the argument.

## 15 Semantics

16 A *numeric-literal* evaluates to either an instance of the class `Integer` or a direct instance of the  
17 class `Float`.

18 NOTE Subclasses of the class `Integer` may be defined as described in 15.2.8.1.

19 An *unsigned-number* of the form *integer-literal* evaluates to an instance of the class `Integer`  
20 whose value is the value of one of the syntactic term sequences in the *integer-literal* production.

21 An *unsigned-number* of the form *float-literal* evaluates to a direct instance of the class `Float`  
22 whose value is the value of one of the syntactic term sequences in the *float-literal* production.

23 A *signed-number* which begins with “+” evaluates to the resulting instance of the *unsigned-*  
24 *number*. A *signed-number* which begins with “-” evaluates to an instance of the class `Integer`  
25 or a direct instance of the class `Float` whose value is the negated value of the resulting instance  
26 of the *unsigned-number*.

27 The value of an *integer-literal*, a *decimal-integer-literal*, a *float-literal*, or a *significand-part* is the  
28 value of one of the syntactic term sequences in their production.

29 The value of a *unprefixed-decimal-integer-literal* is 0 if it is of the form “0”, otherwise the value  
30 of the *unprefixed-decimal-integer-literal* is the value of a sequence of characters, which consist of  
31 a *decimal-digit-except-zero* followed by a sequence of *decimal-digits*, ignoring interleaving “\_”s,  
32 computed using base 10.

33 The value of a *prefixed-decimal-integer-literal* is the value of the *digit-decimal-part*.

34 The value of a *digit-decimal-part* is the value of the sequence of *decimal-digits*, ignoring inter-  
35 leaving “\_”s, computed using base 10.

1 The value of a *binary-integer-literal* is the value of the sequence of *binary-digits*, ignoring inter-  
2 leaving “\_”s, computed using base 2.

3 The value of an *octal-integer-literal* is the value of the sequence of *octal-digits*, ignoring inter-  
4 leaving “\_”s, computed using base 8.

5 The value of a *hexadecimal-integer-literal* is the value of the sequence of *hexadecimal-digits*,  
6 ignoring interleaving “\_”s, computed using base 16. The values of *hexadecimal-digits* **a** (or **A**)  
7 through **f** (or **F**) are 10 through 15, respectively.

8 The value of a *float-literal-without-exponent* is the value of the *unprefixed-decimal-integer-literal*  
9 plus the value of the *digit-decimal-part* times  $10^{-n}$  where  $n$  is the number of *decimal-digits* of  
10 the *digit-decimal-part*.

11 The value of a *float-literal-with-exponent* is the value of the *significand-part* times  $10^n$  where  $n$   
12 is the value of the *exponent-part*.

13 The value of an *exponent-part* is the negative value of the *digit-decimal-part* if “-” occurs,  
14 otherwise, it is the value of the *digit-decimal-part*.

15 See 15.2.8.1 for the range of the value of an instance of the class **Integer**.

16 See 15.2.9.1 for the precision of the value of an instance of the class **Float**.

### 17 **8.7.6.3 String literals**

#### 18 **8.7.6.3.1 General description**

##### 19 **Syntax**

---

```
20 string-literal ::  
21     single-quoted-string  
22     | double-quoted-string  
23     | quoted-non-expanded-literal-string  
24     | quoted-expanded-literal-string  
25     | here-document  
26     | external-command-execution
```

---

##### 27 **Semantics**

28 A *string-literal* evaluates to a direct instance of the class **String**.

29 NOTE Some of the *string-literals* represents a value of an expression (see 8.7.6.3.3), not only the literal  
30 characters of the program text.

#### 31 **8.7.6.3.2 Single quoted strings**

##### 32 **Syntax**

---

```

1  single-quoted-string ::
2      ' single-quoted-string-character* '

3  single-quoted-string-character ::
4      single-quoted-string-non-escaped-character
5      | single-quoted-escape-sequence

6  single-quoted-escape-sequence ::
7      single-escape-character-sequence
8      | single-quoted-string-non-escaped-character-sequence

9  single-escape-character-sequence ::
10     \ single-quoted-string-meta-character

11 single-quoted-string-non-escaped-character-sequence ::
12     \ single-quoted-string-non-escaped-character

13 single-quoted-string-meta-character ::
14     ' | \

15 single-quoted-string-non-escaped-character ::
16     source-character but not single-quoted-string-meta-character

```

---

## 17 Semantics

18 A *single-quoted-string* consists of zero or more characters enclosed by single quotes. The sequence  
19 of *single-quoted-string-characters* within the pair of single quotes represents the content of a  
20 string as it occurs in a program text literally, except for *single-escape-character-sequences*. The  
21 sequence “\” represents “\”. The sequence “\’” represents “’”.

22 NOTE Unlike a *single-escape-character-sequence*, a *single-quoted-string-non-escaped-character-sequence*  
23 represents two characters as it occurs in a program text literally. For example, ‘\a’ represents two  
24 characters \ and a.

25 EXAMPLE ‘\a\’\’ represents a string whose content is “\a\”.

### 26 8.7.6.3.3 Double quoted strings

## 27 Syntax

---

```

28 double-quoted-string ::
29     " double-quoted-string-character* "

30 double-quoted-string-character ::
31     source-character but not ( " | # | \ )
32     | # [lookahead ∉ { $, @, { } ]

```

```

1      | double-escape-sequence
2      | interpolated-character-sequence

3  double-escape-sequence ::
4      simple-escape-sequence
5      | non-escaped-sequence
6      | line-terminator-escape-sequence
7      | octal-escape-sequence
8      | hexadecimal-escape-sequence
9      | control-escape-sequence

10 simple-escape-sequence ::
11     \ double-escaped-character

12 double-escaped-character ::
13     \ | n | t | r | f | v | a | e | b | s

14 non-escaped-sequence ::
15     \ non-escaped-double-quoted-string-character

16 non-escaped-double-quoted-string-character ::
17     source-character but not ( alpha-numeric-character | line-terminator )

18 octal-escape-sequence ::
19     \ octal-digit octal-digit? octal-digit?

20 hexadecimal-escape-sequence ::
21     \ x hexadecimal-digit hexadecimal-digit?

22 control-escape-sequence ::
23     \ ( C- | c ) control-escaped-character

24 control-escaped-character ::
25     double-escape-sequence
26     | ?
27     | source-character but not ( \ | ? )

28 interpolated-character-sequence ::
29     # global-variable-identifier
30     | # class-variable-identifier
31     | # instance-variable-identifier
32     | # { compound-statement }

33 alpha-numeric-character ::
34     uppercase-character
35     | lowercase-character

```

2 **Semantics**

3 A *double-quoted-string* consists of zero or more characters enclosed by double quotes. The se-  
4 quence of *double-quoted-string-characters* within the pair of double quotes represents the content  
5 of a string.

6 Except for a *double-escape-sequence* and an *interpolated-character-sequence*, a *double-quoted-*  
7 *string-character* represents a character as it occurs in a program text.

8 A *simple-escape-sequence* represents a character as shown in Table 1.

**Table 1 – Simple escape sequences**

Escape sequence	Character code
<code>\\</code>	0x5c
<code>\n</code>	0x0a
<code>\t</code>	0x09
<code>\r</code>	0x0d
<code>\f</code>	0x0c
<code>\v</code>	0x0b
<code>\a</code>	0x07
<code>\e</code>	0x1b
<code>\b</code>	0x08
<code>\s</code>	0x20

9 An *octal-escape-sequence* represents a character the code of which is the value of the sequence  
10 of *octal-digits* computed using base 8.

11 A *hexadecimal-escape-sequence* represents a character the code of which is the value of the  
12 sequence of *hexadecimal-digits* computed using base 16.

13 A *non-escaped-sequence* represents an implementation-defined character.

14 A *line-terminator-escape-sequence* is used to break the content of a string into separate lines in  
15 a program text without inserting a *line-terminator* into the string. A *line-terminator-escape-*  
16 *sequence* does not count as a character of the string.

17 A *control-escape-sequence* represents a character the code of which is computed by performing a  
18 bitwise AND operation between 0x9f and the code of the character represented by the *control-*  
19 *escaped-character*, except when the *control-escaped-character* is `?`, in which case, the *control-*  
20 *escape-sequence* represents a character the code of which is 127.

21 An *interpolated-character-sequence* is a part of a *string-literal* which is dynamically evaluated  
22 when the *string-literal* in which it is embedded is evaluated. The value of a *string-literal* which  
23 contains *interpolated-character-sequences* is a direct instance of the class `String` the content of  
24 which is made from the *string-literal* where each occurrence of *interpolated-character-sequence*

1 is replaced by the content of an instance of the class `String` which is the dynamically evaluated  
2 value of the *interpolated-character-sequence*.

3 An *interpolated-character-sequence* is evaluated as follows:

- 4 a) If it is of the form `# global-variable-identifier`, evaluate the *global-variable-identifier* (see  
5 11.5.4.4). Let  $V$  be the resulting value.
- 6 b) If it is of the form `# class-variable-identifier`, evaluate the *class-variable-identifier* (see  
7 11.5.4.5). Let  $V$  be the resulting value.
- 8 c) If it is of the form `# instance-variable-identifier`, evaluate the *instance-variable-identifier*  
9 (see 11.5.4.6). Let  $V$  be the resulting value.
- 10 d) If it is of the form `# { compound-statement }`, evaluate the *compound-statement* (see 10.2).  
11 Let  $V$  be the resulting value.
- 12 e) If  $V$  is an instance of the class `String`, the value of *interpolated-character-sequence* is  $V$ .
- 13 f) Otherwise, invoke the method `to_s` on  $V$  with no arguments. Let  $S$  be the resulting value.
- 14 g) If  $S$  is an instance of the class `String`, the value of *interpolated-character-sequence* is  $S$ .
- 15 h) Otherwise, the behavior is unspecified.

16 EXAMPLE `"1 + 1 = #{1 + 1}"` represents a string whose content is `"1 + 1 = 2"`.

#### 17 8.7.6.3.4 Quoted non-expanded literal strings

##### 18 Syntax

---

19 *quoted-non-expanded-literal-string* ::  
20 `%q non-expanded-delimited-string`

21 *non-expanded-delimited-string* ::  
22 `literal-beginning-delimiter non-expanded-literal-string* literal-ending-delimiter`

23 *non-expanded-literal-string* ::  
24 `non-expanded-literal-character`  
25 `| non-expanded-delimited-string`

26 *non-expanded-literal-character* ::  
27 `non-escaped-literal-character`  
28 `| non-expanded-literal-escape-sequence`

29 *non-escaped-literal-character* ::  
30 `source-character` **but not** `quoted-literal-escape-character`

```

1  non-expanded-literal-escape-sequence ::
2      non-expanded-literal-escape-character-sequence
3      | non-escaped-non-expanded-literal-character-sequence

4  non-expanded-literal-escape-character-sequence ::
5      \ non-expanded-literal-escaped-character

6  non-expanded-literal-escaped-character ::
7      literal-beginning-delimiter
8      | literal-ending-delimiter
9      | \

10 quoted-literal-escape-character ::
11     non-expanded-literal-escaped-character

12 non-escaped-non-expanded-literal-character-sequence ::
13     \ non-escaped-non-expanded-literal-character

14 non-escaped-non-expanded-literal-character ::
15     source-character but not non-expanded-literal-escaped-character

16 literal-beginning-delimiter ::
17     source-character but not alpha-numeric-character

18 literal-ending-delimiter ::
19     source-character but not alpha-numeric-character

```

---

20 All *literal-beginning-delimiters* in a *non-expanded-delimited-string* shall be the same character.  
21 All *literal-ending-delimiters* in a *non-expanded-delimited-string* shall be the same character.

22 If a *literal-beginning-delimiter* is one of the characters on the left in Table 2, the corresponding  
23 *literal-beginning-delimiter* shall be the corresponding character on the right in Table 2. Other-  
24 wise, the *literal-ending-delimiter* shall be the same character as the *literal-beginning-delimiter*.

**Table 2 – Matching *literal-beginning-delimiter* and *literal-ending-delimiter***

<i>literal-beginning-delimiter</i>	<i>literal-ending-delimiter</i>
{	}
(	)
[	]
<	>

25 The *non-expanded-delimited-string* of a *non-expanded-literal-string* in a *quoted-non-expanded-*  
26 *literal-string* applies only when its *literal-beginning-delimiter* is one of the characters on the left  
27 in Table 2.

1 NOTE 1 A *quoted-non-expanded-literal-string* can have nested brackets in regard to the *literal-beginning-*  
2 *delimiter* and the corresponding *literal-ending-delimiter* (e.g., %q[[abc] [def]]). Different brackets than  
3 these two brackets and any escaped brackets are ignored in this nesting. For example, %q[\[abc\]def()  
4 represents a direct instance of the class `String` whose content is “[abc\]def()”. In this case, only [,  
5 ], and \ can be *non-expanded-literal-escaped-characters* because the *literal-beginning-delimiter* and the  
6 corresponding *literal-beginning-delimiter* are [ and ] respectively.

## 7 Semantics

8 The value of a *quoted-non-expanded-literal-string* represents a string whose content is the con-  
9 catenation of the contents represented by the *non-expanded-literal-strings* of the *non-expanded-*  
10 *delimited-string* of the *quoted-non-expanded-literal-string*.

11 The value of a *non-expanded-literal-string* represents the content of a string as it occurs in a  
12 program text literally, except for *non-expanded-literal-escape-character-sequences*.

13 NOTE 1 The content of a string represented by a *non-expanded-literal-string* contains the *literal-*  
14 *beginning-delimiter* and the *literal-ending-delimiter* of a *non-expanded-delimited-string* in the *non-expanded-*  
15 *literal-string*. For example, %q((abc)) represents a direct instance of the class `String` whose content is  
16 “(abc)”.

17 The value of a *non-expanded-literal-escape-character-sequence* represents a character as fol-  
18 lows. The sequence “\\” represents “\”; the sequence “\” *literal-beginning-delimiter*, a *literal-*  
19 *beginning-delimiter*; the sequence “\” *literal-ending-delimiter*, a *literal-ending-delimiter*.

### 20 8.7.6.3.5 Quoted expanded literal strings

#### 21 Syntax

---

22 *quoted-expanded-literal-string* ::  
23 % Q<sup>?</sup> *expanded-delimited-string*

24 *expanded-delimited-string* ::  
25 *literal-beginning-delimiter* *expanded-literal-string*\* *literal-ending-delimiter*

26 *expanded-literal-string* ::  
27 *expanded-literal-character*  
28 | *expanded-delimited-string*

29 *expanded-literal-character* ::  
30 *non-escaped-literal-character* **but not** #  
31 | # [lookahead ∉ { \$, @, { } ]  
32 | *double-escape-sequence*  
33 | *interpolated-character-sequence*

---

34 All *literal-beginning-delimiters* in a *expanded-delimited-string* shall be the same character. All  
35 *literal-ending-delimiters* in a *expanded-delimited-string* shall be the same character.

36 The *literal-ending-delimiter* shall match the *literal-beginning-delimiter* as described in 8.7.6.3.4.

1 The *expanded-delimited-string* of a *expanded-literal-string* in a *quoted-expanded-literal-string* ap-  
2 plies only when its *literal-beginning-delimiter* is one of the characters on the left in 8.7.6.3.4  
3 Table 2.

#### 4 **Semantics**

5 The value of a *quoted-expanded-literal-string* represents a string whose content is the concatena-  
6 tion of the contents represented by the *expanded-literal-strings* of the *expanded-delimited-string*  
7 of the *quoted-expanded-literal-string*.

8 A character in an *expanded-literal-string* other than a *double-escape-sequence* or an *interpolated-*  
9 *character-sequence* represents a character as it occurs in a program text. A *double-escape-*  
10 *sequence* and an *interpolated-character-sequence* represent characters as described in 8.7.6.3.3.

11 NOTE The content of a string represented by a *expanded-literal-string* contains the *literal-beginning-*  
12 *delimiter* and the *literal-ending-delimiter* of a *expanded-delimited-string* in the *expanded-literal-string*.  
13 For example, “%Q({1 + 2})” represents a string whose content is “(3)”.

#### 14 **8.7.6.3.6 Here documents**

#### 15 **Syntax**

---

16 *here-document* ::  
17     *heredoc-start-line* *heredoc-body* *heredoc-end-line*

18 *heredoc-start-line* ::  
19     *heredoc-signifier* *rest-of-line*

20 *heredoc-signifier* ::  
21     << *heredoc-delimiter-specifier*

22 *rest-of-line* ::  
23     *line-content*? *line-terminator*

24 *heredoc-body* ::  
25     *heredoc-body-line*\*

26 *heredoc-body-line* ::  
27     *comment-line* **but not** *heredoc-end-line*

28 *heredoc-delimiter-specifier* ::  
29     -? *heredoc-delimiter*

30 *heredoc-delimiter* ::  
31     *non-quoted-delimiter*  
32     | *single-quoted-delimiter*  
33     | *double-quoted-delimiter*  
34     | *command-quoted-delimiter*

```

1  non-quoted-delimiter ::
2      non-quoted-delimiter-identifier

3  non-quoted-delimiter-identifier ::
4      identifier-character*

5  single-quoted-delimiter ::
6      ' single-quoted-delimiter-identifier '

7  single-quoted-delimiter-identifier ::
8      ( ( source-character source-character? ) but not ( ' | line-terminator ) )*

9  double-quoted-delimiter ::
10     " double-quoted-delimiter-identifier "

11 double-quoted-delimiter-identifier ::
12     ( ( source-character source-character? ) but not ( " | line-terminator ) )*

13 command-quoted-delimiter ::
14     ` command-quoted-delimiter-identifier `

15 command-quoted-delimiter-identifier ::
16     ( ( source-character source-character? ) but not ( ` | line-terminator ) )*

17 heredoc-end-line ::
18     indented-heredoc-end-line
19     | non-indented-heredoc-end-line

20 indented-heredoc-end-line ::
21     [ beginning of a line ] whitespace* heredoc-delimiter-identifier line-terminator

22 non-indented-heredoc-end-line ::
23     [ beginning of a line ] heredoc-delimiter-identifier line-terminator

24 heredoc-delimiter-identifier ::
25     non-quoted-delimiter-identifier
26     | single-quoted-delimiter-identifier
27     | double-quoted-delimiter-identifier
28     | command-quoted-delimiter-identifier

```

---

29 The *heredoc-signifier*, the *heredoc-body*, and the *heredoc-end-line* in a *here-document* are treated  
30 as a unit and considered to be a single token occurring at the place where the *heredoc-signifier*  
31 occurs. The first character of the *rest-of-line* becomes the head of the input after the *here-*  
32 *document* has been processed.

1 The form of a *heredoc-end-line* depends on the presence or absence of the beginning “-” of the  
2 *heredoc-delimiter-specifier*.

3 If the *heredoc-delimiter-specifier* begins with “-”, a line of the form *indented-heredoc-end-line*  
4 is treated as the *heredoc-end-line*, otherwise, a line of the form *non-indented-heredoc-end-line*  
5 is treated as the *heredoc-end-line*. In both forms, the *heredoc-delimiter-identifier* shall be the  
6 same sequence of characters as it occurs in the corresponding part of *heredoc-delimiter*.

7 If the *heredoc-delimiter* is of the form *non-quoted-delimiter*, the *heredoc-delimiter-identifier* shall  
8 be the same sequence of characters as the *non-quoted-delimiter-identifier*; if it is of the form  
9 *single-quoted-delimiter*, the *single-quoted-delimiter-identifier*; if it is of the form of *double-quoted-*  
10 *delimiter*, the *double-quoted-delimiter-identifier*; if it is of the form of *command-quoted-delimiter*,  
11 the *command-quoted-delimiter-identifier*.

## 12 Semantics

13 A *here-document* evaluates to a direct instance of the class **String** or the value of the invocation  
14 of the method ‘.

15 The object to which a *here-document* evaluates is created as follows:

16 a) Create a direct instance *S* of the class **String** from the *heredoc-body*, the content of which  
17 depends on the form of the *heredoc-delimiter* as follows:

- 18 • If *heredoc-delimiter* is of the form *single-quoted-delimiter*, the content of *S* is the se-  
19 quence of *source-characters* of the *heredoc-body*.
- 20 • If *heredoc-delimiter* is in any of the forms *non-quoted-delimiter*, *double-quoted-delimiter*,  
21 or *command-quoted-delimiter*, the content of *S* is the sequence of characters which is  
22 represented by the *heredoc-body* as a sequence of *double-quoted-string-characters* (see  
23 8.7.6.3.3).

24 b) If the *heredoc-delimiter* is not of the form *command-quoted-delimiter*, let *V* be *S*.

25 c) Otherwise, invoke the method ‘ on the current self with the list of arguments which has  
26 only one element *S*. Let *V* be the resulting value of the method invocation.

27 d) *V* is the object to which the *here-document* evaluates.

### 28 8.7.6.3.7 External command execution

#### 29 Syntax

---

30 *external-command-execution* ::  
31     *backquoted-external-command-execution*  
32     | *quoted-external-command-execution*

33 *backquoted-external-command-execution* ::  
34     ‘ *backquoted-external-command-execution-character*\* ‘

1 *backquoted-external-command-execution-character* ::  
2     *source-character* **but not** ( ‘ | # | \ )  
3     | # [lookahead ∉ { \$, @, { } ]  
4     | *double-escape-sequence*  
5     | *interpolated-character-sequence*

6 *quoted-external-command-execution* ::  
7     %x *expanded-delimited-string*

---

8 The *literal-ending-delimiter* shall match the *literal-beginning-delimiter* as described in 8.7.6.3.4.

## 9 Semantics

10 An *external-command-execution* is a form to invoke the method ‘.

11 An *external-command-execution* is evaluated as follows:

- 12 a) If the *external-command-execution* is of the form *backquoted-external-command-execution*,  
13 construct a direct instance *S* of the class **String** whose content is a sequence of characters  
14 represented by *backquoted-external-command-execution-characters*. A *backquoted-external-*  
15 *command-execution-character* other than a *double-escape-sequence* or an *interpolated-character-*  
16 *sequence* represents a character as it occurs in a program text. A *double-escape-sequence*  
17 and an *interpolated-character-sequence* represent characters as described in 8.7.6.3.3.
- 18 b) If the *external-command-execution* is of the form *quoted-external-command-execution*, con-  
19 struct a direct instance *S* of the class **String** by replacing “%x” with “%Q” and evaluating  
20 the resulting *quoted-expanded-literal-string* as described in 8.7.6.3.5.
- 21 c) Invoke the method ‘ on the current self with a list of arguments which has only one element  
22 *S*.
- 23 d) The value of the *external-command-execution* is the resulting value.

## 24 8.7.6.4 Array literals

### 25 Syntax

---

26 *array-literal* ::  
27     *quoted-non-expanded-array-constructor*  
28     | *quoted-expanded-array-constructor*

29 *quoted-non-expanded-array-constructor* ::  
30     %w *literal-beginning-delimiter* *non-expanded-array-content* *literal-ending-delimiter*

31 *non-expanded-array-content* ::  
32     *quoted-array-item-separator-list*? *non-expanded-array-item-list*?  
33     *quoted-array-item-separator-list*?

```

1  non-expanded-array-item-list ::
2      non-expanded-array-item ( quoted-array-item-separator-list non-expanded-array-item )*

3  quoted-array-item-separator-list ::
4      quoted-array-item-separator+

5  quoted-array-item-separator ::
6      whitespace
7      | line-terminator

8  non-expanded-array-item ::
9      non-expanded-array-item-character+

10 non-expanded-array-item-character ::
11     non-escaped-array-character
12     | non-expanded-array-escape-sequence

13 non-escaped-array-character ::
14     non-escaped-literal-character but not quoted-array-item-separator

15 non-expanded-array-escape-sequence ::
16     non-expanded-literal-escape-sequence
17     | \ quoted-array-item-separator

18 quoted-expanded-array-constructor ::
19     %W literal-beginning-delimiter expanded-array-content literal-ending-delimiter

20 expanded-array-content ::
21     quoted-array-item-separator-list? expanded-array-item-list?
22     quoted-array-item-separator-list?

23 expanded-array-item-list ::
24     expanded-array-item ( quoted-array-item-separator-list expanded-array-item )*

25 expanded-array-item ::
26     expanded-array-item-character+

27 expanded-array-item-character ::
28     non-escaped-array-item-character
29     | # [lookahead ∉ { $, @, { }]
30     | expanded-array-escape-sequence
31     | interpolated-character-sequence

32 non-escaped-array-item-character ::
33     source-character but not ( quoted-array-item-separator | \ | # )

```

1 *expanded-array-escape-sequence* ::  
2 *double-escape-sequence*  
3 | \ *quoted-array-item-separator*

---

4 The *literal-ending-delimiter* shall match the *literal-beginning-delimiter* as described in 8.7.6.3.4.

5 If the *literal-beginning-delimiter* is none of the characters on the left in 8.7.6.3.4 Table 2, the  
6 *non-escaped-array-item-character* shall not be the *literal-beginning-delimiter*.

7 If the *literal-beginning-delimiter* is one of the characters on the left in 8.7.6.3.4 Table 2, the  
8 *quoted-non-expanded-array-constructor* or *quoted-expanded-array-constructor* shall satisfy the  
9 following conditions, where *C* is the *quoted-non-expanded-array-constructor* or *quoted-expanded-*  
10 *array-constructor*, *B* is the *literal-beginning-delimiter*, and *E* is the *literal-ending-delimiter* which  
11 corresponds to *B* in 8.7.6.3.4 Table 2, and “the number of *x* in *y*” means the number of *x*  
12 to appear in *y* except appearances in *non-expanded-array-escape-sequences* or *expanded-array-*  
13 *escape-sequences*:

- 14 • The number of *B* in *C* and the number of *E* in *C* are the same.
- 15 • For any substring *S* of *C* which starts from the first *B* and ends before the last *E*, the  
16 number of *B* in *S* is larger than the number of *E* in *S*.

17 NOTE The above conditions are for nested brackets in an *array-literal*. Matching of brackets is ir-  
18 relevant to the structure of the value of an *array-literal*. For example, %w[[ab cd][ef]] represents  
19 ["[ab", "cd][ef]"].

## 20 Semantics

21 An *array-literal* evaluates to a direct instance of the class **Array** as follows:

22 a) A *quoted-non-expanded-array-constructor* is evaluated as follows:

- 23 1) Create an empty direct instance of the class **Array**. Let *A* be the instance.
- 24 2) If *non-expanded-array-item-list* is present, for each *non-expanded-array-item* of the *non-*  
25 *expanded-array-item-list*, take the following steps:
  - 26 i) Create a direct instance *S* of the class **String**, the content of which is represented  
27 by the sequence of *non-expanded-array-item-characters*.

28 A *non-expanded-array-item-character* represents itself, except in the case of a  
29 *non-expanded-array-escape-sequence*. A *non-expanded-array-escape-sequence* rep-  
30 represents a character represented by the *non-expanded-literal-escape-sequence* as de-  
31 scribed in 8.7.6.3.4, except when the *non-expanded-array-escape-sequence* is of the  
32 form \ *quoted-array-item-separator*. A *non-expanded-array-escape-sequence* of the  
33 form \ *quoted-array-item-separator* represents the *quoted-array-item-separator* as  
34 it occurs in a program text literally.

- 35 ii) Append *S* to *A*.

36 3) The value of the *quoted-non-expanded-array-constructor* is *A*.

1 b) A *quoted-expanded-array-constructor* is evaluated as follows:

2 1) Create an empty direct instance of the class `Array`. Let *A* be the instance.

3 2) If *expanded-array-item-list* is present, process each *expanded-array-item* of the *expanded-*  
4 *array-item-list* as follows:

5 i) Create a direct instance *S* of the class `String`, the content of which is represented  
6 by the sequence of *expanded-array-item-characters*.

7 An *expanded-array-item-character* represents itself, except in the case of an *expanded-*  
8 *array-escape-sequence* and an *interpolated-character-sequence*. An *expanded-array-*  
9 *escape-sequence* represents a character represented by the *double-escape-sequence*  
10 as described in 8.7.6.3.3, except when the *expanded-array-escape-sequence* is of  
11 the form `\ quoted-array-item-separator`. An *expanded-array-escape-sequence* of the  
12 form `\ quoted-array-item-separator` represents the *quoted-array-item-separator* as  
13 it occurs in a program text literally. An *interpolated-character-sequence* represents  
14 a sequence of characters as described in 8.7.6.3.3.

15 ii) Append *S* to *A*.

16 3) The value of the *quoted-expanded-array-constructor* is *A*.

## 17 8.7.6.5 Regular expression literals

### 18 Syntax

---

19 *regular-expression-literal* ::  
20 / *regular-expression-body* / *regular-expression-option*\*  
21 | %r *literal-beginning-delimiter* *expanded-literal-string*\*  
22 *literal-ending-delimiter* *regular-expression-option*\*

23 *regular-expression-body* ::  
24 *regular-expression-character*\*

25 *regular-expression-character* ::  
26 *source-character* **but not** ( / | # | \ )  
27 | # [lookahead  $\notin$  { \$, @, { } ]  
28 | *regular-expression-unescaped-sequence*  
29 | *regular-expression-escape-sequence*  
30 | *line-terminator-escape-sequence*  
31 | *interpolated-character-sequence*

32 *regular-expression-unescaped-sequence* ::  
33 \ *regular-expression-unescaped-character*

34 *regular-expression-unescaped-character* ::  
35 *source-character* **but not** ( 0x0d | 0x0a )  
36 | 0x0d [lookahead  $\notin$  { 0x0a } ]

1 *regular-expression-escape-sequence* ::  
2     \*/*

3 *regular-expression-option* ::  
4     *i* | *m*

---

5 Within an *expanded-literal-string* of a *regular-expression-literal*, a *literal-beginning-delimiter* shall  
6 be the same character as the *literal-beginning-delimiter* of the *regular-expression-literal*.

7 The *literal-ending-delimiter* shall match the *literal-beginning-delimiter* as described in 8.7.6.3.4.

## 8 Semantics

9 A *regular-expression-literal* evaluates to a direct instance of the class **Regexp**.

10 The pattern attribute of an instance of the class **Regexp** (see 15.2.15.1) resulting from a *regular-*  
11 *expression-literal* is the string represented by *regular-expression-characters* or *expanded-literal-*  
12 *strings*. The string shall be of the form *pattern* (see 15.2.15.4).

13 A *regular-expression-character* other than a *regular-expression-escape-sequence*, *line-terminator-*  
14 *escape-sequence*, or *interpolated-character-sequence* represents itself as it occurs in a program text  
15 literally. An *expanded-literal-string* other than a *line-terminator-escape-sequence* or *interpolated-*  
16 *character-sequence* represents itself as it occurs in a program text literally.

17 A *regular-expression-escape-sequence* represents the character */*.

18 A *line-terminator-escape-sequence* in a *regular-expression-character* and an *expanded-literal-*  
19 *string* is ignored in the resulting pattern of an instance of the class **Regexp**.

20 An *interpolated-character-sequence* in a *regular-expression-literal* and an *expanded-literal-string*  
21 is evaluated as described in 8.7.6.3.3, and represents a string which is the content of the resulting  
22 instance of the class **String**.

23 A *regular-expression-option* specifies the ignorecase-flag and the multiline-flag attributes of an  
24 instance of the class **Regexp** resulting from a *regular-expression-literal*. If *i* is present in a *regular-*  
25 *expression-option*, the ignorecase-flag attribute of the resulting instance of the class **Regexp** is  
26 set to true. If *m* is present in a *regular-expression-option*, the multiline-flag attribute of the  
27 resulting instance of the class **Regexp** is set to true.

28 The grammar for a pattern of an instance of the class **Regexp** created from a *regular-expression-*  
29 *literal* is described in 15.2.15.4.

## 30 8.7.6.6 Symbol literals

### 31 Syntax

---

32 *symbol* ::  
33     *symbol-literal*  
34     | *dynamic-symbol*

```

1  symbol-literal ::
2      : symbol-name

3  dynamic-symbol ::
4      : single-quoted-string
5      | : double-quoted-string
6      | %s literal-beginning-delimiter non-expanded-literal-string* literal-ending-delimiter

7  symbol-name ::
8      instance-variable-identifier
9      | global-variable-identifier
10     | class-variable-identifier
11     | constant-identifier
12     | local-variable-identifier
13     | method-only-identifier
14     | assignment-like-method-identifier
15     | operator-method-name
16     | keyword

```

---

17 The *single-quoted-string*, *double-quoted-string*, or *non-expanded-literal-string* of the *dynamic-*  
18 *symbol* shall not contain any sequence which represents the character 0x00 in the resulting value  
19 of the *single-quoted-string*, *double-quoted-string*, or *non-expanded-literal-string* as described in  
20 8.7.6.3.2, 8.7.6.3.3, or 8.7.6.3.4.

21 Within a *non-expanded-literal-string*, *literal-beginning-delimiter* shall be the same character as  
22 the *literal-beginning-delimiter* of the *dynamic-symbol*.

23 The *literal-ending-delimiter* shall match the *literal-beginning-delimiter* as described in 8.7.6.3.4.

## 24 Semantics

25 A *symbol* evaluates to a direct instance of the class `Symbol`. A *symbol-literal* evaluates to a direct  
26 instance of the class `Symbol` whose name is the *symbol-name*. A *dynamic-symbol* evaluates to a  
27 direct instance of the class `Symbol` whose name is the content of an instance of the class `String`  
28 which is the value of the *single-quoted-string* (see 8.7.6.3.2), *double-quoted-string* (see 8.7.6.3.3),  
29 or *non-expanded-literal-string* (see 8.7.6.3.4). If the content of the instance of the class `String`  
30 contains the character 0x00, a direct instance of the class `ArgumentError` may be raised.

## 31 9 Scope of variables

### 32 9.1 General description

33 The **scope** of a variable is a set of regions of a program text with which a set of bindings of  
34 variables is associated.

35 Instance variables, constants, and class variables have no scope, and their bindings are searched  
36 depending on values of attributes of execution contexts (see 11.5.4.2, 11.5.4.5, and 11.5.4.6).

## 1 9.2 Scope of local variables

2 A local variable is referred to by a *local-variable-identifier*.

3 Scopes for local variables are introduced by the following program constructs:

- 4 • *program* (see 10.1)
- 5 • *class-body* (see 13.2.2)
- 6 • *module-body* (see 13.1.2)
- 7 • *singleton-class-body* (see 13.4.2)
- 8 • *method-definition* (see 13.3.1) and *singleton-method-definition* (see 13.4.3), for both of which  
9 the scope starts with the *method-parameter-part* and continues up to and including the  
10 *method-body*.
- 11 • *block* (see 11.3.3)

12 Let  $P$  be any of the above program constructs. Let  $S$  be the region of  $P$  excluding all the regions  
13 of any of the above program constructs (except *block*) nested within  $P$ . Then,  $S$  is the **local**  
14 **variable scope** which corresponds to the program construct  $P$ .

15 The scope of a local variable is the local variable scope whose set of local variable bindings  
16 contains the binding of the local variable, which is resolved as described below.

17 Given a *local-variable-identifier* which is a reference to a local variable, the binding of the local  
18 variable is resolved as follows:

- 19 a) Let  $N$  be the *local-variable-identifier*. Let  $B$  be the current set of local variable bindings.
- 20 b) Let  $S$  be the scope of  $B$ .
- 21 c) If a binding with name  $N$  exists in  $B$ , that binding is the resolved binding.
- 22 d) If a binding with name  $N$  does not exist in  $B$ :
  - 23 1) If  $S$  is a local variable scope which corresponds to a *block*:
    - 24 i) If the *local-variable-identifier* occurs as a *left-hand-side* of a *block-parameter-list*,  
25 whether to proceed to the next step or not is implementation-defined.
    - 26 ii) Let new  $B$  be the element immediately below the current  $B$  on  $\llbracket$ local-variable-  
27 bindings $\rrbracket$ , and continue searching for a binding with name  $N$  from Step b).
  - 28 2) Otherwise, a binding is considered not resolved.

## 29 9.3 Scope of global variables

30 The scope of global variables is global in the sense that they are accessible everywhere in a  
31 program. Global variable bindings are created in  $\llbracket$ global-variable-bindings $\rrbracket$ .

# 1 10 Program structure

## 2 10.1 Program

### 3 Syntax

---

4 *program* ::  
5 *compound-statement*

---

6 The program text of a strictly conforming program shall be an element of the set of sequences  
7 of characters represented by the nonterminal symbol *program*. A conforming processor need not  
8 accept programs that include program constructs which cannot be evaluated.

### 9 Semantics

10 A *program* is evaluated as follows:

- 11 a) Push an empty set onto  $\llbracket$ local-variable-bindings $\rrbracket$ .
- 12 b) Evaluate the *compound-statement*.
- 13 c) The value of the *program* is the resulting value.
- 14 d) Restore the execution context by removing the element from the top of  $\llbracket$ local-variable-  
15 bindings $\rrbracket$ .

## 16 10.2 Compound statement

### 17 Syntax

---

18 *compound-statement* ::  
19 *statement-list*? *separator-list*?

20 *statement-list* ::  
21 *statement* ( *separator-list* *statement* )\*

22 *separator-list* ::  
23 *separator*<sup>+</sup>

24 *separator* ::  
25 ;  
26 | [*line-terminator* here]

---

### 27 Semantics

28 A *compound-statement* is evaluated as follows:

- 1 a) If the *statement-list* of the *compound-statement* is omitted, the value of the *compound-*  
2 *statement* is **nil**.
- 3 b) If the *statement-list* of the *compound-statement* is present, evaluate each *statement* of the  
4 *statement-list* in the order it appears in the program text. The value of the *compound-*  
5 *statement* is the value of the last *statement* of the *statement-list*.

## 6 11 Expressions

### 7 11.1 General description

#### 8 Syntax

---

9 *expression* ::  
10 *keyword-logical-expression*

---

11 An *expression* is a program construct which make up a *statement* (see 12). A single *expression*  
12 can be a *statement* as an *expression-statement* (see 12.2).

13 NOTE A difference between an *expression* and a *statement* is that an *expression* is ordinarily used  
14 where its value is required, but a *statement* is ordinarily used where its value is not necessarily required.  
15 However, there are some exceptions. For example, a *jump-expression* (see 11.5.2.4) does not have a value,  
16 and the value of the last *statement* of a *compound-statement* can be used.

#### 17 Semantics

18 See 11.2.2 for *keyword-logical-expressions*.

### 19 11.2 Logical expressions

#### 20 11.2.1 General description

#### 21 Syntax

---

22 *logical-expression* ::=  
23 *logical-NOT-expression*  
24 | *logical-AND-expression*  
25 | *logical-OR-expression*

---

26 Any of *logical-NOT-expression* , *logical-AND-expression*, and *logical-OR-expression* is a concep-  
27 tual name, which is used to organize that of the form using a keyword (e.g., “**not x**”) and that  
28 of the form using an operator (e.g, “**!x**”), because they are syntactically away from each other.

29 See 11.2.3 for *logical-NOT-expressions*. See 11.2.4 for *logical-AND-expressions*. See 11.2.5 for  
30 *logical-OR-expressions*.

## 1 11.2.2 Keyword logical expressions

### 2 Syntax

---

3 *keyword-logical-expression* ::  
4     *keyword-NOT-expression*  
5     | *keyword-AND-expression*  
6     | *keyword-OR-expression*

---

7 See 11.2.3 for *keyword-NOT-expressions*. See 11.2.4 for *keyword-AND-expressions*. See 11.2.5  
8 for *keyword-OR-expressions*.

## 9 11.2.3 Logical NOT expressions

### 10 Syntax

---

11 *logical-NOT-expression* ::=  
12     *keyword-NOT-expression*  
13     | *operator-NOT-expression*

14 *keyword-NOT-expression* ::  
15     *method-invocation-without-parentheses*  
16     | *operator-expression*  
17     | ! *method-invocation-without-parentheses*  
18     | **not** *keyword-NOT-expression*

19 *operator-NOT-expression* ::=  
20     ! ( *method-invocation-without-parentheses* | *unary-expression* )

---

### 21 Semantics

22 A *logical-NOT-expression* is evaluated as follows:

- 23 a) If it is of the form *method-invocation-without-parentheses*, evaluate it as described in 11.3.
- 24 b) If it is of the form *operator-expression*, evaluate it as described in 11.4.
- 25 c) Otherwise:
- 26     1) If it is of the form **not** *keyword-NOT-expression*, evaluate the *keyword-NOT-expression*.  
27         Let *X* be the resulting value.
- 28     2) If it is an *operator-NOT-expression*, evaluate its *method-invocation-without-parentheses*  
29         or *unary-expression*. Let *X* be the resulting value.
- 30     3) If *X* is a trueish object, the value of the *keyword-NOT-expression* or the *operator-*  
31         *NOT-expression* is **false**.

1 4) Otherwise, the value of the *keyword-NOT-expression* or the *operator-NOT-expression*  
2 is **true**.

3 d) If it is a *operator-NOT-expression*, instead of Step c), the *operator-NOT-expression* may be  
4 evaluated as follows:

5 1) Evaluate the *method-invocation-without-parentheses* or the *unary-expression*. Let *V* be  
6 the resulting value.

7 2) Create an empty list of arguments *L*. Invoke the method !@ on *V* with *L* as the list of  
8 arguments. The value of the *operator-NOT-expression* is the resulting value.

9 In this case, the processor shall:

- 10 • include the operator !@ in *operator-method-name*.
- 11 • define an instance method !@ in the class `Object`, one of its superclasses (see 6.5.4), or  
12 a module included in the class `Object`. The method !@ shall not take any arguments  
13 and shall return **true** if the receiver is **false** or **nil**, and shall return **false** otherwise.

#### 14 11.2.4 Logical AND expressions

##### 15 Syntax

---

16 *logical-AND-expression* ::=  
17 *keyword-AND-expression*  
18 | *operator-AND-expression*

19 *keyword-AND-expression* ::  
20 *expression* [no *line-terminator* here] **and** *keyword-NOT-expression*

21 *operator-AND-expression* ::  
22 *equality-expression*  
23 | *operator-AND-expression* [no *line-terminator* here] **&&** *equality-expression*

---

##### 24 Semantics

25 A *logical-AND-expression* is evaluated as follows:

26 a) If the *logical-AND-expression* is a *equality-expression*, evaluate the *equality-expression* as  
27 described in 11.4.4.

28 b) Otherwise:

29 1) Evaluate the *expression* or the *operator-AND-expression*. Let *X* be the resulting value.

30 2) If *X* is a trueish object, evaluate the *keyword-NOT-expression* or *equality-expression*.  
31 Let *Y* be the resulting value. The value of the *keyword-AND-expression* or the *operator-*  
32 *AND-expression* is *Y*.

1       3) Otherwise, the value of the *keyword-AND-expression* or the *operator-AND-expression*  
2       is *X*.

### 3 **11.2.5 Logical OR expressions**

#### 4 **Syntax**

---

5     *logical-OR-expression* ::=  
6         *keyword-OR-expression*  
7         | *operator-OR-expression*

8     *keyword-OR-expression* ::  
9         *expression* [no *line-terminator* here] or *keyword-NOT-expression*

10    *operator-OR-expression* ::  
11        *operator-AND-expression*  
12        | *operator-OR-expression* [no *line-terminator* here] || *operator-AND-expression*

---

#### 13 **Semantics**

14 A *logical-OR-expression* is evaluated as follows:

15 a) If the *logical-OR-expression* is a *operator-AND-expression*, evaluate the *operator-AND-*  
16 *expression* as described in 11.2.4.

17 b) Otherwise:

18     1) Evaluate the *expression* or the *operator-OR-expression*. Let *X* be the resulting value.

19     2) If *X* is a falseish object, evaluate the *keyword-NOT-expression* or the *operator-AND-*  
20 *expression*. Let *Y* be the resulting value. The value of the *keyword-OR-expression* or  
21 *operator-OR-expression* is *Y*.

22     3) Otherwise, the value of the *keyword-OR-expression* or *operator-OR-expression* is *X*.

### 23 **11.3 Method invocation expressions**

#### 24 **11.3.1 General description**

#### 25 **Syntax**

---

26     *method-invocation-expression* ::=  
27         *primary-method-invocation*  
28         | *method-invocation-without-parentheses*  
29         | *local-variable-identifier*

30     *primary-method-invocation* ::  
31         *super-with-optional-argument*

```

1      | indexing-method-invocation
2      | method-only-identifier
3      | method-identifier block
4      | method-identifier
5          [no line-terminator here] [no whitespace here] argument-with-parentheses
6      block?
7      | primary-expression [no line-terminator here] . method-name
8          ( [no line-terminator here] [no whitespace here] argument-with-parentheses )?
9      block?
10     | primary-expression [no line-terminator here] :: method-name
11         [no line-terminator here] [no whitespace here] argument-with-parentheses
12     block?
13     | primary-expression [no line-terminator here] :: method-name-except-constant
14     block?

15     method-identifier ::
16         local-variable-identifier
17     | constant-identifier
18     | method-only-identifier

19     method-name ::
20         method-identifier
21     | operator-method-name
22     | keyword

23     indexing-method-invocation ::
24         primary-expression [no line-terminator here] [no whitespace here]
25         [ indexing-argument-list? ]

26     method-name-except-constant ::
27         method-name but not constant-identifier

28     method-invocation-without-parentheses ::
29         command
30     | chained-command-with-do-block
31     | chained-command-with-do-block ( . | :: ) method-name
32         argument-without-parentheses
33     | return-with-argument
34     | break-with-argument
35     | next-with-argument

36     command ::
37         super-with-argument
38     | yield-with-argument
39     | method-identifier argument-without-parentheses
40     | primary-expression [no line-terminator here] ( . | :: ) method-name
41     argument-without-parentheses

```

```

1  chained-command-with-do-block ::
2      command-with-do-block chained-method-invocation*

3  chained-method-invocation ::
4      ( . | :: ) method-name
5      | ( . | :: ) method-name [no line-terminator here] [no whitespace here]
6      argument-with-parentheses

7  command-with-do-block ::
8      super-with-argument-and-do-block
9      | method-identifier argument-without-parentheses do-block
10     | primary-expression [no line-terminator here]
11     ( . | :: ) method-name argument-without-parentheses do-block

```

---

12 See 11.5.4.7 for *method-invocation-expressions* of the form *local-variable-identifier*.

13 If the *argument-with-parentheses* (see 11.3.2) of a *primary-method-invocation* is present, and the  
14 *argument-list* of the *argument-with-parentheses* is a *block-argument*, the *block* of the *primary-*  
15 *method-invocation* shall be omitted.

16 If the *argument-without-parentheses* of a *command-with-do-block* is present, and the *block-*  
17 *argument* of the *argument-list* of the *argument-without-parentheses* (see 11.3.2) is present, the  
18 *do-block* of the *command-with-do-block* shall be omitted.

## 19 Semantics

20 A *method-invocation-expression* is evaluated as follows:

21 a) A *primary-method-invocation* is evaluated as follows:

22 1) If the *primary-method-invocation* is a *super-with-optional-argument* (see 11.3.4) or an  
23 *indexing-method-invocation*, evaluate it. The value of the *primary-method-invocation*  
24 is the resulting value.

25 2) i) If the *primary-method-invocation* is a *method-only-identifier*, let *O* be the current  
26 self and let *M* be the *method-only-identifier*. Create an empty list of arguments  
27 *L*.

28 ii) If the *method-identifier* of the *primary-method-invocation* is present:

29 I) Let *O* be the current self and let *M* be the *method-identifier*.

30 II) If the *argument-with-parentheses* is present, construct a list of arguments and  
31 a block from the *argument-with-parentheses* as described in 11.3.2. Let *L* be  
32 the resulting list. Let *B* be the resulting block, if any.

33 If the *argument-with-parentheses* is omitted, create an empty list of arguments  
34 *L*.

35 III) If the *block* is present, let *B* be the *block*.

- 1           iii) If “.” of the *primary-method-invocation* is present:
- 2           I) Evaluate the *primary-expression* and let  $O$  be the resulting value. Let  $M$  be  
3           the *method-name*.
- 4           II) If the *argument-with-parentheses* is present, construct a list of arguments and  
5           a block from the *argument-with-parentheses* as described in 11.3.2. Let  $L$  be  
6           the resulting list. Let  $B$  be the resulting block, if any.
- 7           If the *argument-with-parentheses* is omitted, create an empty list of arguments  
8            $L$ .
- 9           III) If the *block* is present, let  $B$  be the *block*.
- 10          iv) If the  $::$  and *method-name* of the *primary-method-invocation* are present:
- 11          I) Evaluate the *primary-expression* and let  $O$  be the resulting value. Let  $M$  be  
12          the *method-name*.
- 13          II) Construct a list of arguments and a block from the *argument-with-parentheses*  
14          as described in 11.3.2. Let  $L$  be the resulting list. Let  $B$  be the resulting block,  
15          if any.
- 16          III) If the *block* is present, let  $B$  be the *block*.
- 17          v) If the  $::$  and *method-name-except-constant* of the *primary-method-invocation* are  
18          present:
- 19          I) Evaluate the *primary-expression* and let  $O$  be the resulting value. Let  $M$  be  
20          the *method-name-except-constant*.
- 21          II) Create an empty list of arguments  $L$ .
- 22          III) If the *block* is present, let  $B$  be the *block*.
- 23          3) Invoke the method  $M$  on  $O$  with  $L$  as the list of arguments and  $B$ , if any, as the block.  
24          (see 13.3.3). The value of the *primary-method-invocation* is the resulting value.
- 25          b) An *indexing-method-invocation* is evaluated as follows:
- 26                  1) Evaluate the *primary-expression*. Let  $O$  be the resulting value.
- 27                  2) If the *indexing-argument-list* is present, construct a list of arguments from the *indexing-*  
28                  *argument-list* as described in 11.3.2. Let  $L$  be the resulting list.
- 29                  3) If the *indexing-argument-list* is omitted, Create an empty list of arguments  $L$ .
- 30                  4) Invoke the method  $[\ ]$  on  $O$  with  $L$  as the list of arguments. The value of the *indexing-*  
31                  *method-invocation* is the resulting value.
- 32          c) A *method-invocation-without-parentheses* is evaluated as follows:

- 1) If the *method-invocation-without-parentheses* is a *command*, evaluate it. The value of the *method-invocation-without-parentheses* is the resulting value.
- 2) If the *method-invocation-without-parentheses* is a *return-with-argument*, *break-with-argument* or *next-with-argument*, evaluate it (see 11.5.2.4). The value of the *method-invocation-without-parentheses* is the resulting value.
- 3) If the *chained-command-with-do-block* of the *method-invocation-without-parentheses* is present:
  - i) Evaluate the *chained-command-with-do-block*. Let  $V$  be the resulting value.
  - ii) If the *method-name* and the *argument-without-parentheses* of the *method-invocation-without-parentheses* are present:
    - I) Let  $M$  be the *method-name*.
    - II) Construct a list of arguments from the *argument-without-parentheses* as described in 11.3.2 and let  $L$  be the resulting list. If the *block-argument* of the *argument-list* of the *argument-without-parentheses* is present, let  $B$  be the *block* to which the *block-argument* corresponds [see 11.3.2 e) 6)].
    - III) Invoke the method  $M$  on  $V$  with  $L$  as the list of arguments and  $B$ , if any, as the block.
    - IV) Replace  $V$  with the resulting value.
  - iii) The value of the *method-invocation-without-parentheses* is  $V$ .
- d) A *command* is evaluated as follows:
  - 1) If the *command* is a *super-with-argument*(see 11.3.4) or a *yield-with-argument* (see 11.3.5), evaluate it. The value of the *command* is the resulting value.
  - 2) Otherwise:
    - i) If the *method-identifier* of the *command* is present:
      - I) If the *method-identifier* is a *local-variable-identifier*, and if the *local-variable-identifier* is considered as a reference to a local variable by the steps in 11.5.4.7.2), and if the *argument-without-parentheses* starts with any of  $\&$ ,  $\ll$ ,  $+$ ,  $-$ ,  $*$ ,  $/$ , and  $\%$ , the behavior is unspecified.
 

NOTE 1 For example, if  $x$  is a reference to a local variable, the behavior of “ $x - 1$ ” is unspecified. The behavior of “ $x - 1$ ” may be the same as an *additive-expression* (see 11.4.4) of the form “ $x - 1$ ”.
      - II) Let  $O$  be the current self and let  $M$  be the *method-identifier*.
      - III) Construct a list of arguments from the *argument-without-parentheses* as described in 11.3.2 and let  $L$  be the resulting list.

1                    If the *block-argument* of the *argument-list* of the *argument-without-parentheses*  
2                    is present, let *B* be the *block* to which the *block-argument* corresponds.

3                    ii) If the *primary-expression*(see 11.5), *method-name*, and *argument-without-parentheses*  
4                    of the *command* are present:

5                    I) Evaluate the *primary-expression*. Let *O* be the resulting value. Let *M* be the  
6                    *method-name*.

7                    II) Construct a list of arguments from the *argument-without-parentheses* as de-  
8                    scribed in 11.3.2 and let *L* be the resulting list.

9                    If the *block-argument* of the *argument-list* of the *argument-without-parentheses*  
10                    is present, let *B* be the *block* to which the *block-argument* corresponds.

11                    iii) Invoke the method *M* on *O* with *L* as the list of arguments and *B*, if any, as the  
12                    block. The value of the *command* is the resulting value.

13 e) A *chained-command-with-do-block* is evaluated as follows:

14                    1) Evaluate the *command-with-do-block* and let *V* be the resulting value.

15                    2) For each *chained-method-invocation*, in the order they appear in the program text, take  
16                    the following steps:

17                    i) Let *M* be the *method-name* of the *chained-method-invocation*.

18                    ii) If the *argument-with-parentheses* is present, construct a list of arguments and a  
19                    block from the *argument-with-parentheses* as described in 11.3.2 and let *L* be the  
20                    resulting list. Let *B* be the resulting block, if any.

21                    If the *argument-with-parentheses* is omitted, create an empty list of arguments *L*.

22                    iii) Invoke the method *M* on *V* with *L* as the list of arguments and *B*, if any, as the  
23                    block.

24                    iv) Replace *V* with the resulting value.

25                    3) The value of the *chained-command-with-do-block* is *V*.

26 f) A *command-with-do-block* is evaluated as follows:

27                    1) If the *command-with-do-block* is a *super-with-argument-and-do-block*, evaluate it. The  
28                    value of the *command-with-do-block* is the resulting value.

29                    2) Otherwise:

30                    i) If the *method-identifier* of the *command-with-do-block* is present:

31                    I) If the *method-identifier* is a *local-variable-identifier*, and if the *local-variable-*  
32                    *identifier* is considered as a reference to a local variable by the steps in

1 11.5.4.7.2), and if the *argument-without-parentheses* starts with any of &, <<,  
2 +, -, \*, /, and %, the behavior is unspecified.

3 NOTE 2 For example, if *x* is a reference to a local variable, the behavior of “*x* -1 do end”  
4 is unspecified.

5 II) Otherwise, let *O* be the current self and let *M* be the *method-identifier*.

6 ii) If the *primary-expression* of the *command-with-do-block* is present, evaluate the  
7 *primary-expression*, and let *O* be the resulting value and let *M* be the *method-*  
8 *name*.

9 iii) Construct a list of arguments from the *argument-without-parentheses* of the *command-*  
10 *with-do-block* and let *L* be the resulting list.

11 iv) Invoke the method *M* on *O* with *L* as the list of arguments and the *do-block* as  
12 the block. The value of the *command-with-do-block* is the resulting value.

### 13 11.3.2 Method arguments

#### 14 Syntax

---

15 *method-argument* ::=  
16 *indexing-argument-list*  
17 | *argument-with-parentheses*  
18 | *argument-without-parentheses*

19 *indexing-argument-list* ::  
20 *command*  
21 | *operator-expression-list* ( [no *line-terminator* here] , )?  
22 | *operator-expression-list* [no *line-terminator* here] , *splatting-argument*  
23 | *association-list* ( [no *line-terminator* here] , )?  
24 | *splatting-argument*

25 *splatting-argument* ::  
26 \* *operator-expression*

27 *operator-expression-list* ::  
28 *operator-expression* ( [no *line-terminator* here] , *operator-expression* )\*

29 *argument-with-parentheses* ::  
30 ( )  
31 | ( *argument-list* )  
32 | ( *operator-expression-list* [no *line-terminator* here] , *chained-command-with-do-*  
33 *block* )  
34 | ( *chained-command-with-do-block* )

35 *argument-without-parentheses* ::  
36 [lookahead ∉ { { } } ] [no *line-terminator* here] *argument-list*

```

1  argument-list ::
2      block-argument
3      | splatting-argument ( , block-argument )?
4      | operator-expression-list [no line-terminator here] , association-list
5          ( [no line-terminator here] , splatting-argument )? ( [no line-terminator
6 here] , block-argument )?
7      | ( operator-expression-list | association-list )
8          ( [no line-terminator here] , splatting-argument )? ( [no line-terminator
9 here] , block-argument )?
10     | command

11  block-argument ::
12     & operator-expression

```

---

13 If an *argument-without-parentheses* starts with a sequence of characters which is any of &, <<,
14 +, -, \*, /, and %:

- 15 • One or more *whitespace* characters shall be present just before the *argument-without-*
16 *parentheses*.
- 17 • No *whitespace* shall be present just after the sequence of characters.

18 NOTE These constraints are necessary to distinguish the sequence of characters from binary operators
19 (see 11.4.4). For example, “x -y” is considered as a *command*. However, “x-y” and “x - y” are not
20 considered as *commands*, but as *additive-expressions*. That is, if x is not a reference to a local variable,
21 the behaviors of “x-y” and “x - y” are the same as “x() - y”.

## 22 Semantics

23 A *method-argument* evaluates to two values: an argument list, and a *block*. These two values
24 are used when the method is invoked. However, a *method-argument* does not have a *block* value
25 depending on evaluation steps.

26 A *method-argument* is evaluated as follows:

27 a) An *indexing-argument-list* is evaluated as follows:

- 28 1) Create an empty list of arguments *L*.
- 29 2) Evaluate the *command*, *operator-expressions* of *operator-expression-lists*, or the *association-*
30 *list* and append their values to *L* in the order they appear in the program text.
- 31 3) If the *splatting-argument* is present, evaluate it, and concatenate the resulting list of
32 arguments to *L*.
- 33 4) The argument list value of *indexing-argument-list* is *L*.

34 b) A *splatting-argument* is evaluated as follows:

- 35 1) Create an empty list of arguments *L*.

- 1       2) Evaluate the *operator-expression*. Let  $V$  be the resulting value.
- 2       3) If  $V$  is not an instance of the class `Array`, the behavior is unspecified.
- 3       4) Append each element of  $V$ , in the indexing order, to  $L$ .
- 4       5) The argument list value of *splatting-argument* is  $L$ .
- 5 c) An *argument-with-parentheses* is evaluated as follows:
  - 6       1) Create an empty list of arguments  $L$ .
  - 7       2) If the *argument-list* is present, evaluate it as described in Step e), and concatenate the  
8       resulting list of arguments to  $L$ . If the *block-argument* of the *argument-list* is present,  
9       the *block* value of the *argument-with-parentheses* is the *block* value of the *argument-list*.
  - 10      3) If the *operator-expression-list* is present, for each *operator-expression* of the *operator-*  
11      *expression-list*, in the order they appear in the program text, take the following steps:
    - 12       i) Evaluate the *operator-expression*. Let  $V$  be the resulting value.
    - 13       ii) Append  $V$  to  $L$ .
  - 14      4) If the *chained-command-with-do-block* is present, evaluate it. Append the resulting  
15      value to  $L$ .
  - 16      5) The argument list value of *argument-with-parentheses* is  $L$ .
- 17 d) An *argument-without-parentheses* is evaluated as follows:
  - 18      1) If the first character of the *argument-without-parentheses* is `(`, the behavior is unspec-  
19      ified.
  - 20      2) Evaluate the *argument-list* as described in Step e).
  - 21      3) Let  $L$  be the resulting list.
- 22 e) An *argument-list* is evaluated as follows:
  - 23      1) Create an empty list of arguments  $L$ .
  - 24      2) If the *command* is present, evaluate it, and append the resulting value to  $L$ .
  - 25      3) If the *operator-expression-list* is present, for each *operator-expression* of the *operator-*  
26      *expression-list*, in the order they appear in the program text, take the following steps:
    - 27       i) Evaluate the *operator-expression*. Let  $V$  be the resulting value.
    - 28       ii) Append  $V$  to  $L$ .
  - 29      4) If the *association-list* is present, evaluate it. Append the resulting value to  $L$ .

- 1       5) If the *splatting-argument* is present, construct a list of arguments from it and concatenate the resulting list to *L*.
- 2
- 3       6) If the *block-argument* is present:
- 4           i) Evaluate the *operator-expression* of the *block-argument*. Let *P* be the resulting value.
- 5
- 6           ii) If *P* is not an instance of the class `Proc`, the behavior is unspecified.
- 7           iii) Otherwise, the *block* value of *argument-list* is the block which *P* represents.
- 8       7) The argument list value of *argument-list* is *L*.

### 9 11.3.3 Blocks

#### 10 Syntax

---

11 *block* ::  
 12       *brace-block*  
 13       | *do-block*

14 *brace-block* ::  
 15       { *block-parameter*<sup>?</sup> *block-body* }

16 *do-block* ::  
 17       do *block-parameter*<sup>?</sup> *block-body* end

18 *block-parameter* ::  
 19       | |  
 20       | ||  
 21       | | *block-parameter-list* |

22 *block-parameter-list* ::  
 23       *left-hand-side*  
 24       | *multiple-left-hand-side*

25 *block-body* ::  
 26       *compound-statement*

---

27 Whether the *left-hand-side* (see 11.4.2.4) in the *block-parameter-list* is allowed to be of the  
 28 following forms is implementation-defined.

- 29 • *constant-identifier*
- 30 • *global-variable-identifier*

- 1 • *instance-variable-identifier*
- 2 • *class-variable-identifier*
- 3 • *primary-expression* [ *indexing-argument-list?* ]
- 4 • *primary-expression* ( . | :: ) ( *local-variable-identifier* | *constant-identifier* )
- 5 • :: *constant-identifier*

6 NOTE Some existing implementations allow some syntactic constructs such as *constant-identifiers* in a  
 7 *block-parameter*. Whether they are allowed is therefore implementation-defined. Future implementations  
 8 should not allow them.

9 Whether the *grouped-left-hand-side* (see 11.4.2.4) of the *multiple-left-hand-side* of the *block-*  
 10 *parameter-list* is allowed to be of the following form is implementation-defined.

- 11 • ( ( *multiple-left-hand-side-item* , )<sup>+</sup> )

## 12 Semantics

13 A *block* is a procedure which is passed to a method invocation.

14 A *block* can be called either by a *yield-expression* (see 11.3.5) or by invoking the method `call`  
 15 on an instance of the class `Proc` which is created by an invocation of the method `new` on the  
 16 class `Proc` to which the block is passed (see 15.2.17.4.3).

17 A *block* can be called with arguments. If a *block* is called by a *yield-expression*, the arguments  
 18 to the *yield-expression* are used as the arguments to the *block* call. If a *block* is called by an  
 19 invocation of the method `call`, the arguments to the method invocation is used as the arguments  
 20 to the *block* call.

21 A *block* is evaluated within the execution context as it exists just before the method invocation to  
 22 which the *block* is passed. However, the changes of variable bindings in `[[local-variable-bindings]]`  
 23 after the *block* is passed to the method invocation affect the execution context. Let  $E_b$  be the  
 24 possibly affected execution context.

25 When a *block* is called, the *block* is evaluated as follows:

- 26 a) Let  $E_o$  be the current execution context. Let  $L$  be the list of arguments passed to the block.
- 27 b) Set the execution context to  $E_b$ .
- 28 c) Push an empty set of local variable bindings onto `[[local-variable-bindings]]`.
- 29 d) If the *block-parameter-list* in the *do-block* or the *brace-block* is present:
  - 30 1) If the *block-parameter-list* is of the form *left-hand-side* or *grouped-left-hand-side*:
    - 31 i) If the length of  $L$  is 0, let  $X$  be `nil`.
    - 32 ii) If the length of  $L$  is 1, let  $X$  be the only element of  $L$ .

- 1           iii) If the length of  $L$  is larger than 1, the result of this step is unspecified.
- 2           iv) If the *block-parameter-list* is of the form *left-hand-side*, evaluate a *single-variable-*  
3 *assignment-expression* (see 11.4.2.2.2)  $E$ , where the *variable* of  $E$  is the *left-hand-*  
4 *side* and the value of the *operator-expression* of  $E$  is  $X$ .
- 5           v) If the *block-parameter-list* is of the form *grouped-left-hand-side*, evaluate a *many-*  
6 *to-many-assignment-statement* (see 11.4.2.4)  $E$ , where the *multiple-left-hand-side*  
7 of  $E$  is the *grouped-left-hand-side* and the value of the *method-invocation-without-*  
8 *parentheses* or *operator-expression* of  $E$  is  $X$ .
- 9       2) If the *block-parameter-list* is of the form *multiple-left-hand-side* and the *multiple-left-*  
10 *hand-side* is not a *grouped-left-hand-side*:
- 11       i) If the length of  $L$  is 1:
- 12           I) If the only element of  $L$  is not an instance of the class `Array`, the result of  
13 this step is unspecified.
- 14           II) Create a list of arguments  $Y$  which contains the elements of  $L$ , preserving  
15 their order.
- 16       ii) If the length of  $L$  is 0 or larger than 1, let  $Y$  be  $L$ .
- 17       iii) Evaluate the *many-to-many-assignment-statement*  $E$  as described in 11.4.2.4, where  
18 the *multiple-left-hand-side* of  $E$  is the *block-parameter-list* and the list of arguments  
19 constructed from the *multiple-right-hand-side* of  $E$  is  $Y$ .
- 20 e) Evaluate the *block-body*. If the evaluation of the *block-body*:
- 21   1) is terminated by a *break-expression*:
- 22       i) If the method invocation with which *block* is passed has already terminated when  
23 the *block* is called:
- 24           I) Let  $S$  be an instance of the class `Symbol` with name `break`.
- 25           II) If the *jump-argument* of the *break-expression* is present, let  $V$  be the value of  
26 the *jump-argument*. Otherwise, let  $V$  be `nil`.
- 27           III) Raise a direct instance of the class `LocalJumpError` which has two instance  
28 variable bindings, one named `@reason` with the value  $S$  and the other named  
29 `@exit_value` with the value  $V$ .
- 30       ii) Otherwise, restore the execution context to  $E_o$  and terminate Step 13.3.3 i) and  
31 take Step 13.3.3 j) of the current method invocation.
- 32       If the *jump-argument* of the *break-expression* is present, the value of the current  
33 method invocation is the value of the *jump-argument*. Otherwise, the value of the  
34 current method invocation is `nil`.
- 35   2) is terminated by a *redo-expression*, repeat Step e).

- 1       3) is terminated by a *next-expression*:
- 2           i) If the *jump-argument* of the *next-expression* is present, let  $V$  be the value of the
- 3                *jump-argument*.
- 4           ii) Otherwise, let  $V$  be **nil**.
- 5       4) is terminated by a *return-expression*, remove the element from the top of  $\llbracket$ local-variable-
- 6                bindings $\rrbracket$ .
- 7       5) is terminates otherwise, let  $V$  be the resulting value of the evaluation of the *block-body*.
- 8 f) Unless Step e) is terminated by a *return-expression*, restore the execution context to  $E_o$ .
- 9 g) The value of calling the *do-block* or the *brace-block* is  $V$ .

#### 10 11.3.4 The super expression

##### 11 Syntax

---

12 *super-expression* ::=

13       *super-with-optional-argument*

14       | *super-with-argument*

15       | *super-with-argument-and-do-block*

16 *super-with-optional-argument* ::

17       **super** ( [no *line-terminator* here] [no *whitespace* here] *argument-with-parentheses* )<sup>?</sup>

18       *block*<sup>?</sup>

19 *super-with-argument* ::

20       **super** *argument-without-parentheses*

21 *super-with-argument-and-do-block* ::

22       **super** *argument-without-parentheses* *do-block*

---

23 The *block-argument* of the *argument-list* of the *argument-without-parentheses* (see 11.3.2) of a

24 *super-with-argument-and-do-block* shall be omitted.

##### 25 Semantics

26 A *super-expression* is evaluated as follows:

- 27 a) If the current self is pushed by a *singleton-class-definition* (see 13.4.2), or an invocation of
- 28 one of the following methods, the behavior is unspecified:
- 29       • the method `class_eval` of the class `Module` (see 15.2.2.4.15)
- 30       • the method `module_eval` of the class `Module` (see 15.2.2.4.35)

- 1       • the method `instance_eval` of the class `Kernel` (see 15.3.1.3.18)
- 2 b) Let  $A$  be an empty list. Let  $B$  be the top of `[[block]]`.
    - 3 1) If the *super-expression* is a *super-with-optional-argument*, and neither the *argument-*  
4 *with-parentheses* nor the *block* is present, construct a list of arguments as follows:
      - 5 i) Let  $M$  be the method which correspond to the current method invocation. Let  $L$   
6 be the *parameter-list* of the *method-parameter-part* of  $M$ . Let  $S$  be the set of local  
7 variable bindings in `[[local-variable-bindings]]` which corresponds to the current  
8 method invocation.
      - 9 ii) If the *mandatory-parameter-list* is present in  $L$ , for each *mandatory-parameter*  $p$ ,  
10 take the following steps:
        - 11 I) Let  $v$  be the value of the binding with name  $p$  in  $S$ .
        - 12 II) Append  $v$  to  $A$ .
      - 13 iii) If the *optional-parameter-list* is present in  $L$ , for each *optional-parameter*  $p$ , take  
14 the following steps:
        - 15 I) Let  $n$  be the *optional-parameter-name* of  $p$ .
        - 16 II) Let  $v$  be the value of the binding with name  $n$  in  $S$ .
        - 17 III) Append  $v$  to  $A$ .
      - 18 iv) If the *array-parameter* is present in  $L$ :
        - 19 I) Let  $n$  be the *array-parameter-name* of the *array-parameter*.
        - 20 II) Let  $v$  be the value of the binding with name  $n$  in  $S$ . Append each element of  
21  $v$ , in the indexing order, to  $A$ .
    - 22 2) If the *super-expression* is a *super-with-optional-argument* with either or both of the  
23 *argument-with-parentheses* and the *block*:
      - 24 i) If the *argument-with-parentheses* is present, construct a list of arguments and a  
25 block as described in 11.3.2. Let  $A$  be the resulting list. Let  $B$  be the resulting  
26 block, if any.
      - 27 ii) If the *block* is present, let  $B$  be the *block*.
    - 28 3) If the *super-expression* is a *super-with-argument*, construct the list of arguments from  
29 the *argument-without-parentheses* as described in 11.3.2. Let  $A$  be the resulting list. If  
30 *block-argument* of the *argument-list* of *argument-without-parentheses* is present, let  $B$   
31 be the *block* constructed from the *block-argument*.
    - 32 4) If the *super-expression* is a *super-with-argument-and-do-block*, construct a list of ar-  
33 guments from the *argument-without-parentheses* as described in 11.3.2. Let  $A$  be the  
34 resulting list. Let  $B$  be the *do-block*.

- 1 c) Determine the method to be invoked as follows:
- 2 1) Let  $C$  be the current class or module. Let  $N$  be the top of `[[defined-method-name]]`.
- 3 2) If  $C$  is an instance of the class `Class`:
- 4 i) Search for a method binding with name  $N$  from Step b) in 13.3.4, assuming that
- 5  $C$  in 13.3.4 to be  $C$ .
- 6 ii) If a binding is found and its value is not `undef` (see 13.1.1), let  $V$  be the value of
- 7 the binding.
- 8 iii) Otherwise:
- 9 I) Add a direct instance of the class `Symbol` with name  $N$  to the head of  $A$ .
- 10 II) Invoke the method `method_missing` (see 15.3.1.3.30) on the current self with
- 11  $A$  as arguments and  $B$  as the block.
- 12 III) Terminate the evaluation of the *super-expression*. The value of the *super-*
- 13 *expression* is the resulting value of the method invocation.
- 14 3) If  $C$  is an instance of the class `Module` and not an instance of the class `Class`:
- 15 i) Let  $M$  be  $C$  and let new  $C$  be the class of the current self.
- 16 ii) Let  $L_m$  be the included module list of  $C$ . Search for  $M$  in  $L_m$ .
- 17 iii) If  $M$  is found in  $L_m$ :
- 18 I) Search for a method binding with name  $N$  in the set of bindings of instance
- 19 methods of each module in  $L_m$ . Examine modules in  $L_m$ , in reverse order,
- 20 from the module just before  $M$  to the first module in  $L_m$ .
- 21 II) If a binding is found and its value is not `undef`, let  $V$  be the value of the
- 22 binding.
- 23 III) If a binding is found and its value is `undef` (see 13.1.1), take the steps from
- 24 c) 2) iii) I) to c) 2) iii) III).
- 25 IV) If a binding is not found and  $C$  has a direct superclass, let  $S$  be the superclass.
- 26 Take Step c) 2), assuming that  $C$  in c) 2) to be  $S$ .
- 27 V) If a binding is not found and  $C$  does not have a direct superclass, take the
- 28 steps from c) 2) iii) I) to c) 2) iii) III).
- 29 iv) Otherwise, let new  $C$  be the direct superclass of  $C$  and repeat from Step c) 3) ii).
- 30 If  $C$  does not have a direct superclass, the behavior is unspecified.
- 31 d) Take steps g), h), i), and j) of 13.3.3, assuming that  $A$ ,  $B$ ,  $M$ ,  $R$ , and  $V$  in 13.3.3 to be  $A$ ,  $B$ ,
- 32  $N$ , the current self, and  $V$  in this subclass respectively. The value of the *super-expression*
- 33 is the resulting value.

## 1 11.3.5 The yield expression

### 2 Syntax

---

3 *yield-expression* ::=  
4     *yield-with-optional-argument*  
5     | *yield-with-argument*

6 *yield-with-optional-argument* ::  
7     *yield-with-parentheses-and-argument*  
8     | *yield-with-parentheses-without-argument*  
9     | **yield**

10 *yield-with-parentheses-and-argument* ::  
11     **yield** [no *line-terminator* here] [no *whitespace* here] ( *argument-list* )

12 *yield-with-parentheses-without-argument* ::  
13     **yield** [no *line-terminator* here] [no *whitespace* here] ( )

14 *yield-with-argument* ::  
15     **yield** *argument-without-parentheses*

---

16 The *block-argument* of the *argument-list* (see 11.3.2) of a *yield-with-parentheses-and-argument*  
17 shall be omitted.

18 The *block-argument* of the *argument-list* of the *argument-without-parentheses* (see 11.3.2) of a  
19 *yield-with-argument* shall be omitted.

### 20 Semantics

21 A *yield-expression* is evaluated as follows:

22 a) Let *B* be the top of `[[block]]`. If *B* is `block-not-given`:

23     1) Let *S* be a direct instance of the class `Symbol` with name `noreason`.

24     2) Let *V* be an implementation-defined value.

25     3) Raise a direct instance of the class `LocalJumpError` which has two instance variable  
26         bindings, one named `@reason` with the value *S* and the other named `@exit_value` with  
27         the value *V*.

28 b) A *yield-with-optional-argument* is evaluated as follows:

29     1) If the *yield-with-optional-argument* is of the form *yield-with-parentheses-and-argument*,  
30         create a list of arguments from the *argument-without-parentheses* as described in 11.3.2.  
31         Let *L* be the list.

- 1        2) If the *yield-with-optional-argument* is of the form *yield-with-parentheses-without-argument*
- 2            or **yield**, create an empty list of argument *L*.
- 3        3) Call *B* with *L* as described in 11.3.3.
- 4        4) The value of the *yield-with-optional-argument* is the value of the block call.
- 5 c) A *yield-with-argument* is evaluated as follows:
  - 6        1) Create a list of arguments from the *argument-without-parentheses* as described in
  - 7            11.3.2. Let *L* be the list.
  - 8        2) Call *B* with *L* as described in 11.3.3.
  - 9        3) The value of the *yield-with-argument* is the value of the block call.

## 10 11.4 Operator expressions

### 11 11.4.1 General description

#### 12 Syntax

---

13 *operator-expression* ::  
14        *assignment-expression*  
15        | *defined?-without-parentheses*  
16        | *conditional-operator-expression*

---

17 See 11.4.2 for *assignment-expressions*.

18 NOTE *assignment-statement* is not an *operator-expression* but a *statement*(see 12.1).

19 See 11.4.3.2 for *defined?-without-parentheses*.

20 NOTE *defined?-with-parentheses* is not an *operator-expression* but a *primary-expression*(see 11.5.1).

21 See 11.5.2.2.5 for *conditional-operator-expressions*.

### 22 11.4.2 Assignments

#### 23 11.4.2.1 General description

#### 24 Syntax

---

25 *assignment* ::=  
26        *assignment-expression*  
27        | *assignment-statement*

28 *assignment-expression* ::  
29        *single-assignment-expression*

1 | *abbreviated-assignment-expression*  
2 | *assignment-with-rescue-modifier*

3 *assignment-statement* ::  
4     *single-assignment-statement*  
5     | *abbreviated-assignment-statement*  
6     | *multiple-assignment-statement*

---

## 7 **Semantics**

8 An *assignment* creates or updates variable bindings, or invokes a method whose name ends with  
9 `=`.

10 Evaluations of *assignment-expressions* and *assignment-statements* are described in the clauses  
11 from 11.4.2.2 to 11.4.2.5.

### 12 **11.4.2.2 Single assignments**

#### 13 **11.4.2.2.1 General description**

#### 14 **Syntax**

---

15 *single-assignment* ::=  
16     *single-assignment-expression*  
17     | *single-assignment-statement*

18 *single-assignment-expression* ::  
19     *single-variable-assignment-expression*  
20     | *scoped-constant-assignment-expression*  
21     | *single-indexing-assignment-expression*  
22     | *single-method-assignment-expression*

23 *single-assignment-statement* ::  
24     *single-variable-assignment-statement*  
25     | *scoped-constant-assignment-statement*  
26     | *single-indexing-assignment-statement*  
27     | *single-method-assignment-statement*

---

#### 28 **11.4.2.2.2 Single variable assignments**

#### 29 **Syntax**

---

30 *single-variable-assignment* ::=  
31     *single-variable-assignment-expression*  
32     | *single-variable-assignment-statement*

1 *single-variable-assignment-expression* ::  
2 *variable* [no *line-terminator* here] = *operator-expression*

3 *single-variable-assignment-statement* ::  
4 *variable* [no *line-terminator* here] = *method-invocation-without-parentheses*

---

## 5 Semantics

6 A *single-variable-assignment* is evaluated as follows:

7 a) Evaluate the *operator-expression* or the *method-invocation-without-parentheses*. Let  $V$  be  
8 the resulting value.

9 b) 1) If the *variable*(see 11.5.4) is a *constant-identifier*:

10 i) Let  $N$  be the *constant-identifier*.

11 ii) If a binding with name  $N$  exists in the set of bindings of constants of the current  
12 class or module, replace the value of the binding with  $V$ .

13 iii) Otherwise, create a variable binding with name  $N$  and value  $V$  in the set of  
14 bindings of constants of the current class or module.

15 2) If the *variable* is a *global-variable-identifier*:

16 i) Let  $N$  be the *global-variable-identifier*.

17 ii) If a binding with name  $N$  exists in  $\llbracket$ global-variable-bindings $\rrbracket$ , replace the value of  
18 the binding with  $V$ .

19 iii) Otherwise, create a variable binding with name  $N$  and value  $V$  in  $\llbracket$ global-variable-  
20 bindings $\rrbracket$ .

21 3) If the *variable* is a *class-variable-identifier*:

22 i) Let  $C$  be the first class or module in the list at the top of  $\llbracket$ class-module-list $\rrbracket$  which  
23 is not a singleton class.

24 Let  $CS$  be the set of classes which consists of  $C$  and all the superclasses of  $C$ . Let  
25  $MS$  be the set of modules which consists of all the modules in the included module  
26 lists of all classes in  $CS$ . Let  $CM$  be the union of  $CS$  and  $MS$ .

27 Let  $N$  be the *class-variable-identifier*.

28 ii) If exactly one of the classes or modules in  $CM$  has a binding with name  $N$  in the  
29 set of bindings of class variables, let  $B$  be that binding.

30 If more than one class or module in  $CM$  has bindings with name  $N$  in the set  
31 of bindings of class variables, choose a binding  $B$  from those bindings in an  
32 implementation-defined way.

- 1           Replace the value of  $B$  with  $V$ .
- 2           iii) If none of the classes or modules in  $CM$  has a binding with name  $N$  in the set of  
3           bindings of class variables, create a variable binding with name  $N$  and value  $V$  in  
4           the set of bindings of class variables of  $C$ .
- 5           4) If the *variable* is an *instance-variable-identifier*:
- 6           i) Let  $N$  be the *instance-variable-identifier*.
- 7           ii) If a binding with name  $N$  exists in the set of bindings of instance variables of the  
8           current self, replace the value of the binding with  $V$ .
- 9           iii) Otherwise, create a variable binding with name  $N$  and value  $V$  in the set of  
10           bindings of instance variables of the current self.
- 11          5) If the *variable* is a *local-variable-identifier*:
- 12          i) Let  $N$  be the *local-variable-identifier*.
- 13          ii) Search for a binding of a local variable with name  $N$  as described in 9.2.
- 14          iii) If a binding is found, replace the value of the binding with  $V$ .
- 15          iv) Otherwise, create a variable binding with name  $N$  and value  $V$  in the current set  
16          of local variable bindings.
- 17   c) The value of the *single-variable-assignment* is  $V$ .

### 18 11.4.2.2.3 Scoped constant assignments

#### 19 Syntax

---

20 *scoped-constant-assignment* ::=

21     *scoped-constant-assignment-expression*

22     | *scoped-constant-assignment-statement*

23 *scoped-constant-assignment-expression* ::

24     *primary-expression* [no *line-terminator* here] [no *whitespace* here] :: *constant-*  
25     *identifier*

26     [no *line-terminator* here] = *operator-expression*

27     | :: *constant-identifier* [no *line-terminator* here] = *operator-expression*

28 *scoped-constant-assignment-statement* ::

29     *primary-expression* [no *line-terminator* here] [no *whitespace* here] :: *constant-*  
30     *identifier*

31     [no *line-terminator* here] = *method-invocation-without-parentheses*

32     | :: *constant-identifier* [no *line-terminator* here] = *method-invocation-without-parentheses*

---

## 1 Semantics

2 A *scoped-constant-assignment* is evaluated as follows:

- 3 a) If the *primary-expression* is present, evaluate it and let  $M$  be the resulting value. Otherwise,  
4 let  $M$  be the class `Object`.
- 5 b) If  $M$  is an instance of the class `Module`:
  - 6 1) Let  $N$  be the *constant-identifier*.
  - 7 2) Evaluate the *operator-expression* or the *method-invocation-without-parentheses*. Let  $V$   
8 be the resulting value.
  - 9 3) If a binding with name  $N$  exists in the set of bindings of constants of  $M$ , replace the  
10 value of the binding with  $V$ .
  - 11 4) Otherwise, create a variable binding with name  $N$  and value  $V$  in the set of bindings  
12 of constants of  $M$ .
  - 13 5) The value of the *scoped-constant-assignment* is  $V$ .
- 14 c) If  $M$  is not an instance of the class `Module`, raise a direct instance of the class `TypeError`.

### 15 11.4.2.2.4 Single indexing assignments

## 16 Syntax

---

17 *single-indexing-assignment* ::=  
18     *single-indexing-assignment-expression*  
19     | *single-indexing-assignment-statement*

20 *single-indexing-assignment-expression* ::  
21     *primary-expression* [no *line-terminator* here] [no *whitespace* here] [ *indexing-*  
22     *argument-list*<sup>?</sup> ]  
23     [no *line-terminator* here] = *operator-expression*

24 *single-indexing-assignment-statement* ::  
25     *primary-expression* [no *line-terminator* here] [no *whitespace* here] [ *indexing-*  
26     *argument-list*<sup>?</sup> ]  
27     [no *line-terminator* here] = *method-invocation-without-parentheses*

---

## 28 Semantics

29 A *single-indexing-assignment* is evaluated as follows:

- 30 a) Evaluate the *primary-expression*. Let  $O$  be the resulting value.
- 31 b) Construct a list of arguments from the *indexing-argument-list* as described in 11.3.2. Let  $L$   
32 be the resulting list.

- 1 c) Evaluate the *operator-expression* or *method-invocation-without-parentheses*. Let  $V$  be the  
 2 resulting value.
- 3 d) Append  $V$  to  $L$ .
- 4 e) Invoke the method  $[]=$  on  $O$  with  $L$  as the list of arguments.
- 5 f) The value of the *single-indexing-assignment* is  $V$ .

#### 6 11.4.2.2.5 Single method assignments

##### 7 Syntax

---

8 *single-method-assignment* ::=  
 9     *single-method-assignment-expression*  
 10    | *single-method-assignment-statement*

11 *single-method-assignment-expression* ::  
 12     *primary-expression* [no *line-terminator* here] ( . | :: ) *local-variable-identifier*  
 13     [no *line-terminator* here] = *operator-expression*  
 14    | *primary-expression* [no *line-terminator* here] . *constant-identifier*  
 15     [no *line-terminator* here] = *operator-expression*

16 *single-method-assignment-statement* ::  
 17     *primary-expression* [no *line-terminator* here] ( . | :: ) *local-variable-identifier*  
 18     [no *line-terminator* here] = *method-invocation-without-parentheses*  
 19    | *primary-expression* [no *line-terminator* here] . *constant-identifier*  
 20     [no *line-terminator* here] = *method-invocation-without-parentheses*

---

##### 21 Semantics

22 A *single-method-assignment* is evaluated as follows:

- 23 a) Evaluate the *primary-expression*. Let  $O$  be the resulting value.
- 24 b) Evaluate the *operator-expression* or *method-invocation-without-parentheses*. Let  $V$  be the  
 25 resulting value.
- 26 c) Let  $M$  be the *local-variable-identifier* or *constant-identifier*. Let  $N$  be the concatenation of  
 27  $M$  and  $=$ .
- 28 d) Invoke the method whose name is  $N$  on  $O$  with a list of arguments which contains only one  
 29 value  $V$ .
- 30 e) The value of the *single-method-assignment* is  $V$ .

### 1 11.4.2.3 Abbreviated assignments

#### 2 11.4.2.3.1 General description

##### 3 Syntax

---

4 *abbreviated-assignment* ::=  
5     *abbreviated-assignment-expression*  
6     | *abbreviated-assignment-statement*

7 *abbreviated-assignment-expression* ::  
8     *abbreviated-variable-assignment-expression*  
9     | *abbreviated-indexing-assignment-expression*  
10    | *abbreviated-method-assignment-expression*

11 *abbreviated-assignment-statement* ::  
12    *abbreviated-variable-assignment-statement*  
13    | *abbreviated-indexing-assignment-statement*  
14    | *abbreviated-method-assignment-statement*

---

#### 15 11.4.2.3.2 Abbreviated variable assignments

##### 16 Syntax

---

17 *abbreviated-variable-assignment* ::=  
18     *abbreviated-variable-assignment-expression*  
19     | *abbreviated-variable-assignment-statement*

20 *abbreviated-variable-assignment-expression* ::  
21     *variable* [no line-terminator here] *assignment-operator operator-expression*

22 *abbreviated-variable-assignment-statement* ::  
23     *variable* [no line-terminator here] *assignment-operator*  
24     *method-invocation-without-parentheses*

---

##### 25 Semantics

26 An *abbreviated-variable-assignment* is evaluated as follows:

- 27 a) Evaluate the *variable* as a variable reference (see 11.5.4). Let *V* be the resulting value.
- 28 b) Evaluate the *operator-expression* or the *method-invocation-without-parentheses*. Let *W* be  
29 the resulting value.
- 30 c) Let *OP* be the *assignment-operator-name* of the *assignment-operator*.

- 1 d) Let  $X$  be the *operator-expression* of the form  $V OP W$ .
- 2 e) Let  $I$  be the *variable* of the *abbreviated-variable-assignment-expression* or the *abbreviated-*  
3 *variable-assignment-statement*.
- 4 f) Evaluate a *single-variable-assignment-expression* (see 11.4.2.2.2) where its *variable* is  $I$  and  
5 the *operator-expression* is  $X$ .
- 6 g) The value of the *abbreviated-variable-assignment* is the resulting value of the evaluation.

### 7 11.4.2.3.3 Abbreviated indexing assignments

#### 8 Syntax

---

9 *abbreviated-indexing-assignment* ::=  
10     *abbreviated-indexing-assignment-expression*  
11     | *abbreviated-indexing-assignment-statement*

12 *abbreviated-indexing-assignment-expression* ::  
13     *primary-expression* [no *line-terminator* here] [no *whitespace* here] [ *indexing-*  
14 *argument-list*? ]  
15     [no *line-terminator* here] *assignment-operator* *operator-expression*

16 *abbreviated-indexing-assignment-statement* ::  
17     *primary-expression* [no *line-terminator* here] [no *whitespace* here] [ *indexing-*  
18 *argument-list*? ]  
19     [no *line-terminator* here] *assignment-operator* *method-invocation-without-parentheses*

---

#### 20 Semantics

- 21 An *abbreviated-indexing-assignment* is evaluated as follows:
- 22 a) Evaluate the *primary-expression*. Let  $O$  be the resulting value.
- 23 b) Construct a list of arguments from the *indexing-argument-list* as described in 11.3.2. Let  $L$   
24 be the resulting list.
- 25 c) Invoke the method  $[]$  on  $O$  with  $L$  as the list of arguments. Let  $V$  be the resulting value.
- 26 d) Evaluate the *operator-expression* or *method-invocation-without-parentheses*. Let  $W$  be the  
27 resulting value.
- 28 e) Let  $OP$  be the *assignment-operator-name* of the *assignment-operator*.
- 29 f) Evaluate the *operator-expression* of the form  $V OP W$ . Let  $X$  be the resulting value.
- 30 g) Append  $X$  to  $L$ .
- 31 h) Invoke the method  $[]=$  on  $O$  with  $L$  as the list of arguments.

1 i) The value of the *abbreviated-indexing-assignment* is  $X$ .

## 2 11.4.2.3.4 Abbreviated method assignments

### 3 Syntax

---

4 *abbreviated-method-assignment* ::=  
5     *abbreviated-method-assignment-expression*  
6     | *abbreviated-method-assignment-statement*

7 *abbreviated-method-assignment-expression* ::  
8     *primary-expression* [no *line-terminator* here] ( . | :: ) *local-variable-identifier*  
9     [no *line-terminator* here] *assignment-operator* *operator-expression*  
10    | *primary-expression* [no *line-terminator* here] . *constant-identifier*  
11    [no *line-terminator* here] *assignment-operator* *operator-expression*

12 *abbreviated-method-assignment-statement* ::  
13     *primary-expression* [no *line-terminator* here] ( . | :: ) *local-variable-identifier*  
14     [no *line-terminator* here] *assignment-operator* *method-invocation-without-parentheses*  
15    | *primary-expression* [no *line-terminator* here] . *constant-identifier*  
16    [no *line-terminator* here] *assignment-operator* *method-invocation-without-parentheses*

---

### 17 Semantics

18 An *abbreviated-method-assignment* is evaluated as follows:

- 19 a) Evaluate the *primary-expression*. Let  $O$  be the resulting value.
- 20 b) Create an empty list of arguments  $L$ . Invoke the method whose name is the *local-variable-identifier* or the *constant-identifier* on  $O$  with  $L$  as the list of arguments. Let  $V$  be the  
21 resulting value.  
22
- 23 c) Evaluate the *operator-expression* or *method-invocation-without-parentheses*. Let  $W$  be the  
24 resulting value.
- 25 d) Let  $OP$  be the *assignment-operator-name* of the *assignment-operator*.
- 26 e) Evaluate the *operator-expression* of the form  $V OP W$ . Let  $X$  be the resulting value.
- 27 f) Let  $M$  be the *local-variable-identifier* or the *constant-identifier*. Let  $N$  be the concatenation  
28 of  $M$  and =.
- 29 g) Invoke the method whose name is  $N$  on  $O$  with a list of arguments which contains only one  
30 value  $X$ .
- 31 h) The value of the *abbreviated-method-assignment* is  $X$ .

## 32 11.4.2.4 Multiple assignments

### 33 Syntax

---

```

1  multiple-assignment-statement ::
2      many-to-one-assignment-statement
3      | one-to-packing-assignment-statement
4      | many-to-many-assignment-statement

5  many-to-one-assignment-statement ::
6      left-hand-side [no line-terminator here] = multiple-right-hand-side

7  one-to-packing-assignment-statement ::
8      packing-left-hand-side [no line-terminator here] =
9      ( method-invocation-without-parentheses | operator-expression )

10 many-to-many-assignment-statement ::
11     multiple-left-hand-side [no line-terminator here] = multiple-right-hand-side
12     | ( multiple-left-hand-side but not packing-left-hand-side )
13         [no line-terminator here] =
14         ( method-invocation-without-parentheses | operator-expression )

15 left-hand-side ::
16     variable
17     | primary-expression [no line-terminator here] [no whitespace here] [ indexing-
18 argument-list? ]
19     | primary-expression [no line-terminator here]
20         ( . | :: ) ( local-variable-identifier | constant-identifier )
21     | :: constant-identifier

22 multiple-left-hand-side ::
23     ( multiple-left-hand-side-item [no line-terminator here] , )+ multiple-left-hand-
24 side-item?
25     | ( multiple-left-hand-side-item [no line-terminator here] , )+ packing-left-hand-side?
26     | packing-left-hand-side
27     | grouped-left-hand-side

28 packing-left-hand-side ::
29     * left-hand-side?

30 grouped-left-hand-side ::
31     ( multiple-left-hand-side )

32 multiple-left-hand-side-item ::
33     left-hand-side
34     | grouped-left-hand-side

35 multiple-right-hand-side ::
36     operator-expression-list ( [no line-terminator here] , splating-right-hand-side )?

```

1 | *splatting-right-hand-side*

2 *splatting-right-hand-side* ::

3 *splatting-argument*

---

#### 4 **Semantics**

5 A *multiple-assignment-statement* is evaluated as follows:

6 a) A *many-to-one-assignment-statement* is evaluated as follows:

7 1) Construct a list of values  $L$  from the *multiple-right-hand-side* as described below.

8 i) If the *operator-expression-list* is present, evaluate its *operator-expressions* in the  
9 order they appear in the program text. Let  $L1$  be a list which contains the resulting  
10 values, preserving their order.

11 ii) If the *operator-expression-list* is omitted, create an empty list of values  $L1$ .

12 iii) If the *splatting-right-hand-side* is present, construct a list of values from its *splatting-*  
13 *argument* as described in 11.3.2 and let  $L2$  be the resulting list.

14 iv) If the *splatting-right-hand-side* is omitted, create an empty list of values  $L2$ .

15 v) The result is the concatenation of  $L1$  and  $L2$ .

16 2) If the length of  $L$  is 0 or 1, let  $A$  be an implementation-defined value.

17 3) If the length of  $L$  is larger than 1, create a direct instance of the class **Array** and store  
18 the elements of  $L$  in it, preserving their order. Let  $A$  be the instance of the class **Array**.

19 4) Evaluate a *single-variable-assignment-expression* (see 11.4.2.2.2) where its *variable* is  
20 the *left-hand-side* and the value of its *operator-expression* is  $A$ .

21 5) The value of the *many-to-one-assignment-statement* is  $A$ .

22 b) A *one-to-packing-assignment-statement* is evaluated as follows:

23 1) Evaluate the *method-invocation-without-parentheses* or the *operator-expression*. Let  $V$   
24 be the resulting value.

25 2) If  $V$  is an instance of the class **Array**, let  $A$  be a a new direct instance of the class  
26 **Array** which contains only one element  $V$  itself, or all the elements of  $V$  in the same  
27 order in  $V$ . Which is chosen is implementation-defined.

28 3) If  $V$  is not an instance of the class **Array**, create a direct instance  $A$  of the class **Array**  
29 which contains only one value  $V$ .

30 4) If the *left-hand-side* of the *packing-left-hand-side* is present, evaluate a *single-variable-*  
31 *assignment-expression* (see 11.4.2.2.2) where its *variable* is the *left-hand-side* and the  
32 value of the *operator-expression* is  $A$ . Otherwise, skip this step.

- 1        5) The value of the *one-to-packing-assignment-statement* is  $A$ .
- 2 c) A *many-to-many-assignment-statement* is evaluated as follows:
- 3        1) If the *multiple-right-hand-side* is present, construct a list of values from it [see a) 1)]  
4            and let  $R$  be the resulting list.
- 5        2) If the *multiple-right-hand-side* is omitted:
- 6            i) Evaluate the *method-invocation-without-parentheses* or the *operator-expression*.  
7                Let  $V$  be the resulting value.
- 8            ii) If  $V$  is not an instance of the class **Array**, the behavior is unspecified.
- 9            iii) Create a list of arguments  $R$  which contains all the elements of  $V$ , preserving their  
10                order.
- 11        3) i) Create an empty list of variables  $L$ .
- 12            ii) For each *multiple-left-hand-side-item*, in the order they appear in the program  
13                text, append the *left-hand-side* or the *grouped-left-hand-side* of the *multiple-left-*  
14                *hand-side-item* to  $L$ .
- 15            iii) If the *packing-left-hand-side* of the *multiple-left-hand-side* is present, append it to  
16                 $L$ .
- 17            iv) If the *multiple-left-hand-side* is a *grouped-left-hand-side*, append the *grouped-left-*  
18                *hand-side* to  $L$ .
- 19        4) For each element  $L_i$  of  $L$ , in the same order in  $L$ , take the following steps:
- 20            i) Let  $i$  be the index of  $L_i$  within  $L$ . Let  $N_R$  be the number of elements of  $R$ .
- 21            ii) If  $L_i$  is a *left-hand-side*:
- 22                I) If  $i$  is larger than  $N_R$ , let  $V$  be **nil**.
- 23                II) Otherwise, let  $V$  be the  $i$ th element of  $R$ .
- 24                III) Evaluate the *single-variable-assignment* of the form  $L_i = V$ .
- 25            iii) If  $L_i$  is a *packing-left-hand-side* and its *left-hand-side* is present:
- 26                I) If  $i$  is larger than  $N_R$ , create an empty direct instance of the class **Array**. Let  
27                     $A$  be the instance.
- 28                II) Otherwise, create a direct instance of the class **Array** which contains elements  
29                    in  $R$  whose index is equal to, or larger than  $i$ , in the same order they are stored  
30                    in  $R$ . Let  $A$  be the instance.
- 31                III) Evaluate a *single-variable-assignment-expression* (see 11.4.2.2.2) where its  
32                    *variable* is the *left-hand-side* and the value of the *operator-expression* is  $A$ .

- 1           iv) If  $L_i$  is a *grouped-left-hand-side*:
- 2               I) If  $i$  is larger than  $N_R$ , let  $V$  be **nil**.
- 3               II) Otherwise, let  $V$  be the  $i$ th element of  $R$ .
- 4               III) Evaluate a *many-to-many-assignment-statement* where its *multiple-left-hand-side* is the *multiple-left-hand-side* of the *grouped-left-hand-side* and its *multiple-right-hand-side* is  $V$ .

#### 7 11.4.2.5 Assignments with rescue modifiers

##### 8 Syntax

---

9     *assignment-with-rescue-modifier* ::=  
10         *left-hand-side* [no *line-terminator* here] =  
11         *operator-expression*<sub>1</sub> [no *line-terminator* here] **rescue** *operator-expression*<sub>2</sub>

---

##### 12 Semantics

13 An *assignment-with-rescue-modifier* is evaluated as follows:

- 14 a) Evaluate the *operator-expression*<sub>1</sub>. Let  $V$  be the resulting value.
- 15 b) If an exception is raised and not handled during the evaluation of the *operator-expression*<sub>1</sub>,  
16 and if the exception is an instance of the class **StandardError**, evaluate the *operator-*  
17 *expression*<sub>2</sub> and replace  $V$  with the resulting value.
- 18 c) Evaluate a *single-variable-assignment-expression* (see 11.4.2.2.2) where its *variable* is the  
19 *left-hand-side* and the value of the *operator-expression* is  $V$ . The value of the *assignment-*  
20 *with-rescue-modifier* is the resulting value of the evaluation.

#### 21 11.4.3 Unary operator expressions

##### 22 11.4.3.1 General description

##### 23 Syntax

---

24     *unary-operator-expression* ::=  
25         *unary-minus-expression*  
26         | *unary-expression*

27     *unary-minus-expression* ::=  
28         *power-expression*  
29         | - *power-expression*

30     *unary-expression* ::=  
31         *primary-expression*

1 |  $\sim$  *unary-expression*<sub>1</sub>  
2 | + *unary-expression*<sub>2</sub>  
3 | ! *unary-expression*<sub>3</sub>

---

#### 4 **Semantics**

5 A *unary-operator-expression* is evaluated as follows:

6 a) A *unary-minus-expression* of the form *power-expression* is evaluated as described in 11.4.4  
7 e).

8 b) A *unary-minus-expression* of the form - *power-expression* is evaluated as follows:

9 1) Evaluate the *power-expression*. Let  $X$  be the resulting value.

10 2) Create an empty list of arguments  $L$ . Invoke the method -@ on  $X$  with  $L$  as the list of  
11 arguments. The value of the *unary-expression* is the resulting value of the invocation.

12 c) A *unary-expression* of the form  $\sim$  *unary-expression*<sub>1</sub> is evaluated as follows:

13 1) Evaluate the *unary-expression*<sub>1</sub>. Let  $X$  be the resulting value.

14 2) Create an empty list of arguments  $L$ . Invoke the method  $\sim$  on  $X$  with  $L$  as the list of  
15 arguments. The value of the *unary-expression* is the resulting value of the invocation.

16 d) A *unary-expression* of the form + *unary-expression*<sub>2</sub> is evaluated as follows:

17 1) Evaluate the *unary-expression*<sub>2</sub>. Let  $X$  be the resulting value.

18 2) Create an empty list of arguments  $L$ . Invoke the method +@ on  $X$  with  $L$  as the list of  
19 arguments. The value of the *unary-expression*<sub>2</sub> is the resulting value of the invocation.

20 e) A *unary-expression* of the form ! *unary-expression*<sub>3</sub> is evaluated as described in 11.2.

#### 21 **11.4.3.2 The defined? expression**

#### 22 **Syntax**

---

23 *defined?-expression* ::=  
24 *defined?-with-parentheses*  
25 | *defined?-without-parentheses*

26 *defined?-with-parentheses* ::=  
27 **defined?** ( *expression* )

28 *defined?-without-parentheses* ::=  
29 **defined?** *operator-expression*

---

## 1 Semantics

2 A *defined?-expression* is evaluated as follows:

3 a) Let  $E$  be the *expression* of the *defined?-with-parentheses* or the *operator-expression* of the  
4 *defined?-without-parentheses*.

5 b) If  $E$  is a *constant-identifier*:

6 1) Search for a binding of a constant with name  $E$  with the same evaluation steps for  
7 *constant-identifier* as described in 11.5.4.2. However, a direct instance of the class  
8 `NameError` shall not be raised when a binding is not found.

9 2) If a binding is found, the value of the *defined?-expression* is an implementation-defined  
10 value, which shall be a trueish object.

11 3) Otherwise, the value of the *defined?-expression* is **nil**.

12 c) If  $E$  is a *global-variable-identifier*:

13 1) If a binding with name  $E$  exists in `[[global-variable-bindings]]`, the value of the *defined?-*  
14 *expression* is an implementation-defined value, which shall be a trueish object.

15 2) Otherwise, the value of the *defined?-expression* is **nil**.

16 d) If  $E$  is a *class-variable-identifier*:

17 1) Let  $C$  be the current class or module. Let  $CS$  be the set of classes which consists of  $C$   
18 and all the superclasses of  $C$ . Let  $MS$  be the set of modules which consists of all the  
19 modules in the included module lists of all classes in  $CS$ . Let  $CM$  be the union of  $CS$   
20 and  $MS$ .

21 2) If any of the classes or modules in  $CM$  has a binding with name  $E$  in the set of bindings  
22 of class variables, the value of the *defined?-expression* is an implementation-defined  
23 value, which shall be a trueish object.

24 3) Otherwise, the value of the *defined?-expression* is **nil**.

25 e) If  $E$  is an *instance-variable-identifier*:

26 1) If a binding with name  $E$  exists in the set of bindings of instance variables of the  
27 current self, the value of the *defined?-expression* is an implementation-defined value,  
28 which shall be a trueish object.

29 2) Otherwise, the value of the *defined?-expression* is **nil**.

30 f) If  $E$  is a *local-variable-identifier*:

31 1) If the *local-variable-identifier* is a reference to a local variable (see 11.5.4.7.2), the value  
32 of the *defined?-expression* is an implementation-defined value, which shall be a trueish  
33 object.

- 1       2) Otherwise, search for a method binding with name  $E$ , starting from the current class  
 2       or module as described in 13.3.4.
- 3           i) If the binding is found and its value is not undef, the value of the *defined?-*  
 4           *expression* is an implementation-defined value, which shall be a trueish object.
- 5           ii) Otherwise, the value of the *defined?-expression* is **nil**.
- 6 g) Otherwise, the value of the *defined?-expression* is implementation-defined.

#### 7 11.4.4 Binary operator expressions

#### 8 Syntax

---

9       *binary-operator-expression* ::=  
 10       *equality-expression*

11       *equality-expression* ::  
 12       *relational-expression*  
 13       | *relational-expression* [no line-terminator here] <=> *relational-expression*  
 14       | *relational-expression* [no line-terminator here] == *relational-expression*  
 15       | *relational-expression* [no line-terminator here] === *relational-expression*  
 16       | *relational-expression* [no line-terminator here] != *relational-expression*  
 17       | *relational-expression* [no line-terminator here] =~ *relational-expression*  
 18       | *relational-expression* [no line-terminator here] !~ *relational-expression*

19       *relational-expression* ::  
 20       *bitwise-OR-expression*  
 21       | *relational-expression* [no line-terminator here] > *bitwise-OR-expression*  
 22       | *relational-expression* [no line-terminator here] >= *bitwise-OR-expression*  
 23       | *relational-expression* [no line-terminator here] < *bitwise-OR-expression*  
 24       | *relational-expression* [no line-terminator here] <= *bitwise-OR-expression*

25       *bitwise-OR-expression* ::  
 26       *bitwise-AND-expression*  
 27       | *bitwise-OR-expression* [no line-terminator here] | *bitwise-AND-expression*  
 28       | *bitwise-OR-expression* [no line-terminator here] ^ *bitwise-AND-expression*

29       *bitwise-AND-expression* ::  
 30       *bitwise-shift-expression*  
 31       | *bitwise-AND-expression* [no line-terminator here] & *bitwise-shift-expression*

32       *bitwise-shift-expression* ::  
 33       *additive-expression*  
 34       | *bitwise-shift-expression* [no line-terminator here] << *additive-expression*  
 35       | *bitwise-shift-expression* [no line-terminator here] >> *additive-expression*

```

1  additive-expression ::
2      multiplicative-expression
3      | additive-expression [no line-terminator here] + multiplicative-expression
4      | additive-expression [no line-terminator here] - multiplicative-expression

5  multiplicative-expression ::
6      unary-minus-expression
7      | multiplicative-expression [no line-terminator here] * unary-minus-expression
8      | multiplicative-expression [no line-terminator here] / unary-minus-expression
9      | multiplicative-expression [no line-terminator here] % unary-minus-expression

10 power-expression ::
11     unary-expression
12     | unary-expression [no line-terminator here] ** power-expression

13 binary-operator ::=
14     <=> | == | != | === | =~ | !~ | > | >= | < | <=
15     | | | ^ | & | << | >> | + | - | * | / | % | **

```

---

16 If there is a *whitespace* character just before any of the following operators, there shall be one  
17 or more *whitespace* characters just after the operator.

- 18 • & of a *bitwise-AND-expression*
- 19 • << of a *bitwise-shift-expression*
- 20 • + of a *additive-expression*
- 21 • - of a *additive-expression*
- 22 • \* of a *multiplicative-expression*
- 23 • / of a *multiplicative-expression*
- 24 • % of a *multiplicative-expression*

25 NOTE This constraint is necessary to distinguish binary operators from leading sequences of characters  
26 of *argument-without-parentheses* (see 11.3.2).

## 27 Semantics

28 An *equality-expression* is evaluated as follows:

- 29 a) If the *equality-expression* is of the form  $x \neq y$ , take the following steps:
  - 30 1) Evaluate  $x$ . Let  $X$  be the resulting value.
  - 31 2) Evaluate  $y$ . Let  $Y$  be the resulting value.

1        3) Invoke the method `==` on  $X$  with  $Y$  as an argument. If the resulting value is a trueish  
2        object, the value of the *equality-expression* is **false**. Otherwise, the value of the *equality-*  
3        *expression* is **true**.

4 b) The steps in Step f) may be taken instead of Step a).

5 In this case, the following conditions shall be satisfied:

- 6        • The operator `!=` is included in *operator-method-name*.
- 7        • An instance method `!=` is defined in the class `Object`, one of its superclasses, or a  
8        module included in the class `Object`. The method `!=` shall take one argument and  
9        shall return the value of the *equality-expression* in Step a) 3), where let  $X$  and  $Y$  be  
10       the receiver and the argument, respectively.

11 c) If the *equality-expression* is of the form  $x !\sim y$ , take the following steps:

12       1) Evaluate  $x$ . Let  $X$  be the resulting value.

13       2) Evaluate  $y$ . Let  $Y$  be the resulting value.

14       3) Invoke the method `=~` on  $X$  with  $Y$  as an argument. If the resulting value is a trueish  
15       object, the value of the *equality-expression* is **false**. Otherwise, the value of the *equality-*  
16       *expression* is **true**.

17 d) The steps in Step f) may be taken instead of Step c). In this case, the following conditions  
18 shall be satisfied:

- 19       • The operator `!~` is included in *operator-method-name*.
- 20       • An instance method `!~` is defined in the class `Object`, one of its superclasses, or a  
21       module included in the class `Object`. The method `!~` shall take one argument and  
22       shall return the value of the *equality-expression* in Step c) 3), where let  $X$  and  $Y$  be  
23       the receiver and the argument, respectively.

24 e) If the *equality-expression* is an *unary-minus-expression* and not a *power-expression*, evalu-  
25       ate it as described in 11.4.3. If the *equality-expression* is an *unary-minus-expression* and  
26       a *power-expression*, evaluate the *power-expression* by taking the following steps and the  
27       resulting value is the value of the *equality-expression*.

28       1) If the *power-expression* is a *unary-expression*, evaluate it as described in 11.4.3 and the  
29       resulting value is the value of the *power-expression*.

30       2) If the *power-expression* is a *power-expression* of the form *unary-expression* **\*\*** *power-*  
31       *expression*:

32       i) If the *unary-expression* is of the form `- unsigned-number`:

33           I) Evaluate the *unsigned-number* and let  $X$  be the resulting value.

34           II) Evaluate the *power-expression*<sub>2</sub> and let  $Y$  be the resulting value.

- 1           III) Invoke the method whose name is “\*\*” on  $X$  with  $Y$  as an argument. Let  $Z$   
2           be the resulting value.
- 3           IV) Invoke the method whose name is “-@” on  $Z$  with no arguments. The value  
4           of the *equality-expression* is the resulting value of the invocation.
- 5           ii) Otherwise:
- 6           I) Evaluate the *unary-expression* and let  $X$  be the resulting value.
- 7           II) Evaluate the *power-expression* and let  $Y$  be the resulting value.
- 8           III) Invoke the method whose name is “\*\*” on  $X$  with  $Y$  as an argument. The  
9           value of the *power-expression* is the resulting value.
- 10 f) Otherwise, for the *equality-expression* of the form  $x$  *binary-operator*  $y$ , take the following  
11 steps:
- 12       1) Evaluate  $x$ . Let  $X$  be the resulting value.
- 13       2) Evaluate  $y$ . Let  $Y$  be the resulting value.
- 14       3) Invoke the method whose name is the *binary-operator* on  $X$  with  $Y$  as an argument.  
15       The value of the *equality-expression* is the resulting value of the invocation.

## 16 11.5 Primary expressions

### 17 11.5.1 General description

#### 18 Syntax

---

19 *primary-expression* ::  
20     *class-definition*  
21     | *singleton-class-definition*  
22     | *module-definition*  
23     | *method-definition*  
24     | *singleton-method-definition*  
25     | *yield-with-optional-argument*  
26     | *if-expression*  
27     | *unless-expression*  
28     | *case-expression*  
29     | *while-expression*  
30     | *until-expression*  
31     | *for-expression*  
32     | *return-without-argument*  
33     | *break-without-argument*  
34     | *next-without-argument*  
35     | *redo-expression*  
36     | *retry-expression*  
37     | *begin-expression*  
38     | *grouping-expression*

1 | *variable-reference*  
2 | *scoped-constant-reference*  
3 | *array-constructor*  
4 | *hash-constructor*  
5 | *literal*  
6 | *defined?-with-parentheses*  
7 | *primary-method-invocation*

---

## 8 **Semantics**

9 See 13.2.2 for *class-definitions*.

10 See 13.4.2 for *singleton-class-definitions*.

11 See 13.1.2 for *module-definitions*.

12 See 13.3.1 for *method-definitions*.

13 See 13.4.3 for *singleton-method-definitions*.

14 See 11.3.5 for *yield-with-optional-arguments*.

15 See 8.7.6 for *literals*.

16 See 11.4.3.2 for *defined?-with-parentheses*.

17 See 11.3 for *primary-method-invocations*.

## 18 **11.5.2 Control structures**

### 19 **11.5.2.1 General description**

#### 20 **Syntax**

---

21 *control-structure ::=*  
22     *conditional-expression*  
23     | *iteration-expression*  
24     | *jump-expression*  
25     | *begin-expression*  
26

---

### 27 **11.5.2.2 Conditional expressions**

#### 28 **11.5.2.2.1 General description**

#### 29 **Syntax**

---

30 *conditional-expression ::=*  
31     *if-expression*

1 | *unless-expression*  
2 | *case-expression*  
3 | *conditional-operator-expression*  
4

---

## 5 11.5.2.2.2 The if expression

### 6 Syntax

---

7 *if-expression* ::  
8     **if** *expression* *then-clause* *elsif-clause*\* *else-clause*? **end**

9 *then-clause* ::  
10     *separator* *compound-statement*  
11     | *separator*? **then** *compound-statement*

12 *else-clause* ::  
13     **else** *compound-statement*

14 *elsif-clause* ::  
15     **elsif** *expression* *then-clause*

---

### 16 Semantics

17 The *if-expression* is evaluated as follows:

- 18 a) Evaluate *expression*. Let  $V$  be the resulting value.
- 19 b) If  $V$  is a trueish object, evaluate the *compound-statement* of the *then-clause*. The value of  
20 the *if-expression* is the resulting value. In this case, *elsif-clauses* and the *else-clause*, if any,  
21 are not evaluated.
- 22 c) If  $V$  is a falseish object, and if there is no *elsif-clause* and no *else-clause*, then the value of  
23 the *if-expression* is **nil**.
- 24 d) If  $V$  is a falseish object, and if there is no *elsif-clause* but there is an *else-clause*, then  
25 evaluate the *compound-statement* of the *else-clause*. The value of the *if-expression* is the  
26 resulting value.
- 27 e) If  $V$  is a falseish object, and if there are one or more *elsif-clauses*, evaluate the sequence of  
28 *elsif-clauses* as follows:
- 29 1) Evaluate the *expression* of each *elsif-clause* in the order they appear in the program  
30 text, until there is an *elsif-clause* for which *expression* evaluates to a trueish object.  
31 Let  $T$  be this *elsif-clause*.

- 1        2) If  $T$  exists, evaluate the *compound-statement* of its *then-clause*. The value of the *if-expression* is the resulting value. Other *elsif-clauses* and an *else-clause* following  $T$ , if  
2        *expression* is the resulting value. Other *elsif-clauses* and an *else-clause* following  $T$ , if  
3        any, are not evaluated.
- 4        3) If  $T$  does not exist, and if there is an *else-clause*, then evaluate the *compound-statement*  
5        of the *else-clause*. The value of the *if-expression* is the resulting value.
- 6        4) If  $T$  does not exist, and if there is no *else-clause*, then the value of the *if-expression* is  
7        **nil**.

### 8    11.5.2.2.3    The unless expression

#### 9    Syntax

---

10    *unless-expression* ::  
11        **unless** *expression then-clause else-clause?* **end**

---

#### 12   Semantics

13    The *unless-expression* is evaluated as follows:

- 14    a) Evaluate the *expression*. Let  $V$  be the resulting value.
- 15    b) If  $V$  is a falseish object, evaluate the *compound-statement* of the *then-clause*. The value  
16        of the *unless-expression* is the resulting value. In this case, the *else-clause*, if any, is not  
17        evaluated.
- 18    c) If  $V$  is a trueish object, and if there is no *else-clause*, then the value of the *unless-expression*  
19        is **nil**.
- 20    d) If  $V$  is a trueish object, and if there is an *else-clause*, then evaluate the *compound-statement*  
21        of the *else-clause*. The value of the *unless-expression* is the resulting value.

### 22   11.5.2.2.4    The case expression

#### 23   Syntax

---

24    *case-expression* ::  
25        *case-expression-with-expression*  
26        | *case-expression-without-expression*

27    *case-expression-with-expression* ::  
28        **case** *expression separator-list?* *when-clause*<sup>+</sup> *else-clause?* **end**

29    *case-expression-without-expression* ::  
30        **case** *separator-list?* *when-clause*<sup>+</sup> *else-clause?* **end**

```

1  when-clause ::
2      when when-argument then-clause

3  when-argument ::
4      operator-expression-list ( [no line-terminator here] , splattling-argument )?
5      | splattling-argument

```

---

## 6 Semantics

7 A *case-expression* is evaluated as follows:

- 8 a) If the *case-expression* is a *case-expression-with-expression*, evaluate the *expression*. Let  $V$ 
 9 be the resulting value.
- 10 b) The meaning of the phrase “ $O$  is matching” in Step c) is defined as follows:
  - 11 1) If the *case-expression* is a *case-expression-with-expression*, invoke the method `===` on
 12  $O$  with a list of arguments which contains only one value  $V$ .  $O$  is matching if and only
 13 if the resulting value is a trueish object.
  - 14 2) If the *case-expression* is a *case-expression-without-expression*,  $O$  is matching if and only
 15 if  $O$  is a trueish object.
- 16 c) Take the following steps:
  - 17 1) Search the *when-clauses* in the order they appear in the program text for a matching
 18 *when-clause* as follows:
    - 19 i) If the *operator-expression-list* of the *when-argument* is present:
      - 20 I) For each of its *operator-expressions*, evaluate it and test if the resulting value
 21 is matching.
      - 22 II) If a matching value is found, other *operator-expressions*, if any, are not eval-
 23 uated.
    - 24 ii) If no matching value is found, and the *splattling-argument*(see 11.3.2) is present:
      - 25 I) Construct a list of values from it as described in 11.3.2. For each element of
 26 the resulting list, in the same order in the list, test if it is matching.
      - 27 II) If a matching value is found, other values, if any, are not evaluated.
    - 28 iii) A *when-clause* is considered to be matching if and only if a matching value is found
 29 in its *when-argument*. Later *when-clauses*, if any, are not tested in this case.
  - 30 2) If one of the *when-clauses* is matching, evaluate the *compound-statement* of the *then-*
 31 *clause* of this *when-clause*. The value of the *case-expression* is the resulting value.

- 1        3) If none of the *when-clauses* is matching, and if there is an *else-clause*, then evaluate  
2        the *compound-statement* of the *else-clause*. The value of the *case-expression* is the  
3        resulting value.
- 4        4) Otherwise, the value of the *case-expression* is **nil**.

### 5    11.5.2.2.5    Conditional operator expression

#### 6    Syntax

---

7        *conditional-operator-expression* ::=  
8            *range-constructor*  
9            | *range-constructor* [no *line-terminator* here] ? *operator-expression*<sub>1</sub> [no *line-*  
10        *terminator* here] : *operator-expression*<sub>2</sub>

---

#### 11   Semantics

12    A *conditional-operator-expression* of the form *range-constructor* ? *operator-expression*<sub>1</sub> : *operator-*  
13    *expression*<sub>2</sub> is evaluated as follows:

- 14    a) Evaluate the *range-constructor*.
- 15    b) If the resulting value is a trueish object, evaluate the *operator-expression*<sub>1</sub>. The value of  
16    the *conditional-operator-expression* is the resulting value of the evaluation.
- 17    c) Otherwise, evaluate the *operator-expression*<sub>2</sub>. The value of the *conditional-operator-expression*  
18    is the resulting value of the evaluation.

### 19   11.5.2.3    Iteration expressions

#### 20   11.5.2.3.1    General description

#### 21   Syntax

---

22        *iteration-expression* ::=  
23            *while-expression*  
24            | *until-expression*  
25            | *for-expression*  
26            | *while-modifier-statement*  
27            | *until-modifier-statement*

---

28    Each *iteration-expression* has a **condition expression** and a **body**.

29    The condition expression of an *iteration-expression* is the *iteration-expression*'s part evaluated to  
30    determine the condition of the iteration of the *iteration-expression*. The condition expression of a  
31    *while-expression* (see 11.5.2.3.2), *until-expression* (see 11.5.2.3.3), *for-expression* (see 11.5.2.3.4),  
32    *while-modifier-statement* (see 12.5) or *until-modifier-statement* (see 12.6) is its *expression*.

1 The body of an *iteration-expression* is the *iteration-expression*'s part evaluated iteratively. The  
2 body of a *while-expression*, *until-expression*, or *for-expression* is its *compound-statement*. The  
3 body of a *while-modifier-statement* or *until-modifier-statement* is its *statement*.

4 See 12.5 for *while-modifier-statements*.

5 See 12.6 for *until-modifier-statements*.

## 6 11.5.2.3.2 The while expression

### 7 Syntax

---

8 *while-expression* ::  
9     **while** *expression do-clause* **end**

10 *do-clause* ::  
11     *separator compound-statement*  
12     | [no *line-terminator* here] **do** *compound-statement*

---

### 13 Semantics

14 A *while-expression* is evaluated as follows:

15 a) Evaluate the *expression*, and take the following steps:

16     1) If the evaluation of the *expression* is terminated by a *break-expression* (see 11.5.2.4.3),  
17         terminate the evaluation of the *while-expression*.

18         If the *jump-argument* of the *break-expression* is present, the value of the *while-expression*  
19         is the value of the *jump-argument*. Otherwise, the value of the *while-expression* is **nil**.

20     2) If the evaluation of the *expression* is terminated by a *next-expression* (see 11.5.2.4.4)  
21         or *redo-expression* (see 11.5.2.4.5), continue processing from the beginning of Step a).

22     3) Otherwise, let *V* be the resulting value of the *expression*.

23 b) If *V* is a falseish object, terminate the evaluation of the *while-expression*. The value of the  
24 *while-expression* is **nil**.

25 c) If *V* is a trueish object, evaluate the *compound-statement* of the *do-clause*, and take the  
26 following steps:

27     1) If the evaluation of the *compound-statement* is terminated by a *break-expression*, ter-  
28         minate the evaluation of the *while-expression*.

29         If the *jump-argument* of the *break-expression* is present, the value of the *while-expression*  
30         is the value of the *jump-argument*. Otherwise, the value of the *while-expression* is **nil**.

31     2) If the evaluation of the *compound-statement* is terminated by a *next-expression*, con-  
32         tinue processing from Step a).

- 1        3) If the evaluation of the *compound-statement* is terminated by a *redo-expression*, con-  
2        tinue processing from Step c).
- 3        4) Otherwise, continue processing from Step a).

#### 4    11.5.2.3.3    The until expression

#### 5    Syntax

---

6        *until-expression* ::  
7        **until** *expression do-clause* **end**

---

#### 8    Semantics

9    An *until-expression* is evaluated as follows:

- 10 a) Evaluate the *expression*, and take the following steps:
- 11        1) If the evaluation of the *expression* is terminated by a *break-expression* (see 11.5.2.4.3),  
12        terminate the evaluation of the *until-expression*.
- 13        If the *jump-argument* of the *break-expression* is present, the value of the *until-expression*  
14        is the value of the *jump-argument*. Otherwise, the value of the *until-expression* is **nil**.
- 15        2) If the evaluation of the *expression* is terminated by a *next-expression* (see 11.5.2.4.4)  
16        or *redo-expression* (see 11.5.2.4.5), continue processing from the beginning of Step a).
- 17        3) Otherwise, let *V* be the resulting value of the *expression*.
- 18 b) If *V* is a trueish object, terminate the evaluation of the *until-expression*. The value of the  
19        *until-expression* is **nil**.
- 20 c) If *V* is a falseish object, evaluate the *compound-statement* of the *do-clause*, and take the  
21        following steps:
- 22        1) If the evaluation of the *compound-statement* is terminated by a *break-expression*, ter-  
23        minate the evaluation of the *until-expression*.
- 24        If the *jump-argument* of the *break-expression* is present, the value of the *until-expression*  
25        is the value of the *jump-argument*. Otherwise, the value of the *until-expression* is **nil**.
- 26        2) If the evaluation of the *compound-statement* is terminated by a *next-expression*, con-  
27        tinue processing from Step a).
- 28        3) If the evaluation of the *compound-statement* is terminated by a *redo-expression*, con-  
29        tinue processing from Step c).
- 30        4) Otherwise, continue processing from Step a).

#### 1 11.5.2.3.4 The for expression

##### 2 Syntax

---

3 *for-expression* ::  
4     **for** *for-variable* [no *line-terminator* here] **in** *expression do-clause* **end**

5 *for-variable* ::  
6     *left-hand-side*  
7     | *multiple-left-hand-side*

---

##### 8 Semantics

9 A *for-expression* is evaluated as follows:

- 10 a) Evaluate the *expression*. If the evaluation of the *expression* is terminated by a *break-expression*, *next-expression*, or *redo-expression*, the behavior is unspecified. Otherwise, let  
11 *O* be the resulting value.  
12
- 13 b) Let *E* be the *primary-method-invocation* of the form *primary-expression* [no *line-terminator*  
14 here] . **each do** | *block-parameter-list* | *block-body* **end**, where the value of the *primary-*  
15 *expression* is *O*, the *block-parameter-list* is the *for-variable*, the *block-body* is the *compound-*  
16 *statement* of the *do-clause*.
- 17 Evaluate *E*; however, if a block whose *block-body* is the *compound-statement* of the *do-clause*  
18 of the *for-expression* is called during this evaluation, the steps in 11.3.3 except the Step c)  
19 and the Step e) 4) shall be taken for the evaluation of this call.
- 20 c) The value of the *for-expression* is the resulting value of the evaluation.

#### 21 11.5.2.4 Jump expressions

##### 22 11.5.2.4.1 General description

##### 23 Syntax

---

24 *jump-expression* ::=  
25     *return-expression*  
26     | *break-expression*  
27     | *next-expression*  
28     | *redo-expression*  
29     | *retry-expression*

---

##### 30 Semantics

31 *jump-expressions* are used to terminate the evaluation of a *method-body*, a *block-body*, the body  
32 of an *iteration-expression*, or the *compound-statement*<sub>2</sub> of a *rescue-clause*. The evaluation of the

1 program construct terminated by a *jump-expression* and the evaluations of program constructs  
2 in the program construct which are under evaluation when the evaluation of the *jump-expression*  
3 has been started are terminated in the middle of the evaluation steps, and have no resulting  
4 values.

5 In this document, the **current block** or the **current iteration-expression** refers to the fol-  
6 lowing:

- 7 a) If the current method invocation does not exist, the *block* or *iteration-expression* whose  
8 evaluation has been started most recently among *blocks* and *iteration-expressions* which  
9 are under evaluation.
- 10 b) If the current method invocation exists, the *block* or *iteration-expression* whose evaluation  
11 has been started most recently among *blocks* and *iteration-expressions* which are under  
12 evaluation and whose evaluation has been started during the evaluation of the current  
13 method invocation.

14 In the both cases, the current *block* or the current *iteration-expression* does not exist if such a  
15 *block* or *iteration-expression* does not exist.

#### 16 11.5.2.4.2 The return expression

##### 17 Syntax

---

18 *return-expression* ::=  
19     *return-without-argument*  
20     | *return-with-argument*

21 *return-without-argument* ::  
22     **return**

23 *return-with-argument* ::  
24     **return** *jump-argument*

25 *jump-argument* ::  
26     [no *line-terminator* here] *argument-list*

---

27 The *block-argument* of the *argument-list* (see 11.3.2) of a *jump-argument* shall be omitted.

##### 28 Semantics

29 *return-expressions* and *jump-arguments* are evaluated as follows:

- 30 a) A *return-expression* is evaluated as follows:
  - 31 1) Let *M* be the *method-body* which corresponds to the current method invocation. If  
32 such an invocation does not exist, or has already terminated:
    - 33 i) Let *S* be a direct instance of the class `Symbol` with name `return`.

- 1           ii) If the *jump-argument* of the *return-expression* is present, let  $V$  be the value of the  
2           *jump-argument*. Otherwise, let  $V$  be **nil**.
- 3           iii) Raise a direct instance of the class `LocalJumpError` which has two instance vari-  
4           able bindings, one named `@reason` with the value  $S$  and the other named `@exit_value`  
5           with the value  $V$ .
- 6           2) Evaluate the *jump-argument*, if any, as described in Step b).
- 7           3) If there are *block-bodys* which include the *return-expression* and are included in  $M$ ,  
8           terminate the evaluations of such *block-bodys*, from innermost to outermost (see 11.3.3).
- 9           4) Terminate the evaluation of  $M$  (see 13.3.3).
- 10       b) A *jump-argument* is evaluated as follows:
- 11       1) If the *jump-argument* is a *splatting-argument*:
- 12           i) Construct a list of values from the *splatting-argument* as described in 11.3.2 and  
13           let  $L$  be the resulting list.
- 14           ii) If the length of  $L$  is 0 or 1, the value of the *jump-argument* is an implementation-  
15           defined value.
- 16           iii) If the length of  $L$  is larger than 1, create a direct instance of the class `Array`  
17           and store the elements of  $L$  in it, preserving their order. The value of the *jump-*  
18           *argument* is the instance.
- 19       2) Otherwise:
- 20           i) Construct a list of values from the *argument-list* as described in 11.3.2 and let  $L$   
21           be the resulting list.
- 22           ii) If the length of  $L$  is 1, the value of the *jump-argument* is the only element of  $L$ .
- 23           iii) If the length of  $L$  is larger than 1, create a direct instance of the class `Array`  
24           and store the elements of  $L$  in it, preserving their order. The value of the *jump-*  
25           *argument* is the instance of the class `Array`.

#### 26 11.5.2.4.3 The break expression

##### 27 Syntax

---

28 *break-expression* ::=  
29     *break-without-argument*  
30     | *break-with-argument*

31 *break-without-argument* ::  
32     **break**

1 *break-with-argument* ::  
2 **break** *jump-argument*

---

### 3 **Semantics**

4 A *break-expression* is evaluated as follows:

- 5 a) Evaluate the *jump-argument*, if any, as described in 11.5.2.4.2 b).
- 6 b) If the current *block* is present, terminate the evaluation of the *block-body* of the current  
7 *block* (see 11.3.3).
- 8 c) If the current *iteration-expression* is present, terminate the evaluation of the condition  
9 expression of the current *iteration-expression* (see 11.5.2.3) when the *break-expression* is in  
10 the condition expression, or terminate the body of the current *iteration-expression* when  
11 the *break-expression* is in the body.
- 12 d) If the current *block* or the current *iteration-expression* is not present:
  - 13 1) Let *S* be a direct instance of the class **Symbol** with name **break**.
  - 14 2) If the *jump-argument* of the *break-expression* is present, let *V* be the value of the  
15 *jump-argument*. Otherwise, let *V* be **nil**.
  - 16 3) Raise a direct instance of the class **LocalJumpError** which has two instance variable  
17 bindings, one named **@reason** with the value *S* and the other named **@exit\_value** with  
18 the value *V*.

#### 19 **11.5.2.4.4 The next expression**

### 20 **Syntax**

---

21 *next-expression* ::=  
22 *next-without-argument*  
23 | *next-with-argument*

24 *next-without-argument* ::  
25 **next**

26 *next-with-argument* ::  
27 **next** *jump-argument*

---

### 28 **Semantics**

29 A *next-expression* is evaluated as follows:

- 30 a) Evaluate the *jump-argument*, if any, as described in 11.5.2.4.2 b).

- 1 b) If the current *block* is present, terminate the evaluation of the *block-body* of the current  
2 *block* (see 11.3.3).
- 3 c) If the current *iteration-expression* is present, terminate the evaluation of the condition  
4 expression of the current *iteration-expression* (see 11.5.2.3) when the *next-expression* is in  
5 the condition expression, or terminate the body of the current *iteration-expression* when  
6 the *next-expression* is in the body.
- 7 d) If the current *block* or the current *iteration-expression* is not present:
- 8 1) Let *S* be a direct instance of the class `Symbol` with name `next`.
- 9 2) If the *jump-argument* of the *next-expression* is present, let *V* be the value of the *jump-*  
10 *argument*. Otherwise, let *V* be `nil`.
- 11 3) Raise a direct instance of the class `LocalJumpError` which has two instance variable  
12 bindings, one named `@reason` with the value *S* and the other named `@exit_value` with  
13 the value *V*.

#### 14 11.5.2.4.5 The redo expression

##### 15 Syntax

---

16 *redo-expression* ::  
17 `redo`

---

##### 18 Semantics

19 A *redo-expression* is evaluated as follows:

- 20 a) If the current *block* is present, terminate the evaluation of the *block-body* of the current  
21 *block* (see 11.3.3).
- 22 b) If the current *iteration-expression* is present, terminate the evaluation of the condition  
23 expression of the current *iteration-expression* (see 11.5.2.3) when the *redo-expression* is in  
24 the condition expression, or terminate the body of the current *iteration-expression* when  
25 the *redo-expression* is in the body.
- 26 c) If the current *block* or the current *iteration-expression* is not present:
- 27 1) Let *S* be a direct instance of the class `Symbol` with name `redo`.
- 28 2) Raise a direct instance of the class `LocalJumpError` which has two instance variable  
29 bindings, one named `@reason` with the value *S* and the other named `@exit_value` with  
30 the value `nil`.

#### 31 11.5.2.4.6 The retry expression

##### 32 Syntax

---

1 *retry-expression* ::  
2 **retry**

---

### 3 **Semantics**

4 A *retry-expression* is evaluated as follows:

- 5 a) If the current method invocation (see 13.3.3) exists, let *M* be the *method-body* which cor-  
6 responds to the current method invocation. Otherwise, let *M* be the *program* (see 10.1).
- 7 b) Let *E* be the innermost *rescue-clause* in *M* which encloses the *retry-expression*. If such a  
8 *rescue-clause* does not exist, the behavior is unspecified.
- 9 c) Terminate the evaluation of the *compound-statement* of the *then-clause* of *E* (see 11.5.2.5).

### 10 **11.5.2.5 The begin expression**

#### 11 **Syntax**

---

12 *begin-expression* ::  
13 **begin** *body-statement* **end**

14 *body-statement* ::  
15 *compound-statement* *rescue-clause*\* *else-clause*? *ensure-clause*?

16 *rescue-clause* ::  
17 **rescue** [no *line-terminator* here] *exception-class-list*?  
18 *exception-variable-assignment*? *then-clause*

19 *exception-class-list* ::  
20 *operator-expression*  
21 | *multiple-right-hand-side*

22 *exception-variable-assignment* ::  
23 => *left-hand-side*

24 *ensure-clause* ::  
25 **ensure** *compound-statement*

---

### 26 **Semantics**

27 The value of a *begin-expression* is the resulting value of the *body-statement*.

28 A *body-statement* is evaluated as follows:

- 1 a) Evaluate the *compound-statement* of the *body-statement*.
- 2 b) If no exception is raised, or all the raised exceptions are handled during Step a):
  - 3 1) If the *else-clause* is present, evaluate the *else-clause* as described in 11.5.2.2.2.
  - 4 2) If the *ensure-clause* is present, evaluate its *compound-statement*. The value of the
  - 5 *ensure-clause* is the value of this evaluation.
  - 6 3) If both the *else-clause* and the *ensure-clause* are present, the value of the *body-statement*
  - 7 is the value of the *ensure-clause*. If only one of these clauses is present, the value of
  - 8 the *body-statement* is the value of the clause.
  - 9 4) If neither the *else-clause* nor the *ensure-clause* is present, the value of the *body-*
  - 10 *statement* is the value of its *compound-statement*.
- 11 c) If an exception is raised and not handled during Step a), test each *rescue-clause*, if any, in
- 12 the order it occurs in the program text. The test determines whether the *rescue-clause* can
- 13 handle the exception as follows:
  - 14 1) Let  $E$  be the exception raised.
  - 15 2) If the *exception-class-list* is omitted in the *rescue-clause*, and if  $E$  is an instance of the
  - 16 class `StandardError`, the *rescue-clause* handles  $E$ .
  - 17 3) If the *exception-class-list* of the *rescue-clause* is present:
    - 18 i) If the *exception-class-list* is of the form *operator-expression*, evaluate the *operator-*
    - 19 *expression*. Create an empty list of values, and append the value of the *operator-*
    - 20 *expression* to the list.
    - 21 ii) If the *exception-class-list* is of the form *multiple-right-hand-side*, construct a list
    - 22 of values from the *multiple-right-hand-side* (see 11.4.2.4).
    - 23 iii) Let  $L$  be the list created by evaluating the *exception-class-list* as above. For each
    - 24 element  $D$  of  $L$ :
      - 25 I) If  $D$  is neither the class `Exception` nor a subclass of the class `Exception`,
      - 26 raise a direct instance of the class `TypeError`.
      - 27 II) If  $E$  is an instance of  $D$ , the *rescue-clause* handles  $E$ . In this case, any re-
      - 28 maining *rescue-clauses* in the *body-statement* are not tested.
  - 29 d) If a *rescue-clause*  $R$  which can handle  $E$  is found:
    - 30 1) If the *exception-variable-assignment* of  $R$  is present, evaluate it in the same way as
    - 31 a *multiple-assignment-statement* of the form *left-hand-side* = *multiple-right-hand-side*
    - 32 where the value of *multiple-right-hand-side* is  $E$ .
    - 33 2) Evaluate the *compound-statement* of the *then-clause* of  $R$ . If this evaluation is termi-
    - 34 nated by a *retry-expression*, continue processing from Step a). Otherwise, let  $V$  be the
    - 35 value of this evaluation.

- 1        3) If the *ensure-clause* is present, evaluate it. The value of the *body-statement* is the value  
 2        of the *ensure-clause*.
- 3        4) If the *ensure-clause* is omitted, the value of the *body-statement* is *V*.
- 4 e) If no *rescue-clause* is present or if a *rescue-clause* which can handle *E* is not found:
- 5        1) If the *ensure-clause* is present, evaluate it.
- 6        2) The value of the *body-statement* is unspecified.

7 The *ensure-clause* of a *body-statement*, if any, is always evaluated, even when the evaluation of  
 8 *body-statement* is terminated by a *jump-expression*.

### 9 11.5.3 Grouping expression

#### 10 Syntax

---

11 *grouping-expression* ::  
 12        ( *expression* )  
 13        | ( *compound-statement* )

---

#### 14 Semantics

15 A *grouping-expression* is evaluated as follows:

- 16 a) Evaluate the *expression* or the *compound-statement*.
- 17 b) The value of the *grouping-expression* is the resulting value.

### 18 11.5.4 Variable references

#### 19 11.5.4.1 General description

#### 20 Syntax

---

21 *variable-reference* ::  
 22        *variable*  
 23        | *pseudo-variable*

24 *variable* ::  
 25        *constant-identifier*  
 26        | *global-variable-identifier*  
 27        | *class-variable-identifier*  
 28        | *instance-variable-identifier*  
 29        | *local-variable-identifier*

30 *scoped-constant-reference* ::  
 31        *primary-expression* [no *line-terminator* here] [no *whitespace* here] :: *constant-*

1        *identifier*  
2        | :: *constant-identifier*

---

3 See from 11.5.4.2 to 11.5.4.7 for *variable* and *scoped-constant-references*.

4 See 11.5.4.8 for *pseudo-variables*.

#### 5 **11.5.4.2 Constants**

6 A *constant-identifier* is evaluated as follows:

7 a) Let  $N$  be the *constant-identifier*.

8 b) Search for a binding of a constant with name  $N$  as described below.

9        As soon as the binding is found in any of the following steps, the evaluation of the *constant-*  
10 *identifier* is terminated and the value of the *constant-identifier* is the value of the binding  
11 found.

12 c) Let  $L$  be the top of `[[class-module-list]]`. Search for a binding of a constant with name  $N$  in  
13 each element of  $L$  from start to end, including the first element, which is the current class  
14 or module, but except for the last element, which is the class `Object`.

15 d) If a binding is not found, let  $C$  be the current class or module.

16        Let  $L$  be the included module list of  $C$ . Search each element of  $L$  in the reverse order for a  
17 binding of a constant with name  $N$ .

18 e) If the binding is not found:

19        1) If  $C$  is an instance of the class `Class`:

20            i) If  $C$  does not have a direct superclass, create a direct instance of the class `Symbol`  
21            with name  $N$ , and let  $R$  be that instance. Invoke the method `const_missing` on  
22            the current class or module with  $R$  as the only argument.

23            ii) Let  $S$  be the direct superclass of  $C$ .

24            iii) Search for a binding of a constant with name  $N$  in  $S$ .

25            iv) If the binding is not found, let  $L$  be the included module list of  $S$  and search each  
26            element of  $L$  in the reverse order for a binding of a constant with name  $N$ .

27            v) If the binding is not found, let  $C$  be the direct superclass of  $S$ . Continue processing  
28            from Step e) 1) i).

29        2) If  $C$  is not an instance of the class `Class`:

30            i) Search for a binding of a constant with name  $N$  in the class `Object`.

- 1           ii) If the binding is not found, let  $L$  be the included module list of the class `Object`  
2           and search each element of  $L$  in the reverse order for a binding of a constant with  
3           name  $N$ .
- 4           iii) If the binding is not found, create a direct instance of the class `Symbol` with name  
5            $N$ , and let  $R$  be that instance. Invoke the method `const_missing` on the current  
6           class or module with  $R$  as the only argument.

#### 7 **11.5.4.3 Scoped constants**

8 A *scoped-constant-reference* is evaluated as follows:

- 9 a) If the *primary-expression* is present, evaluate it and let  $C$  be the resulting value. Otherwise,  
10 let  $C$  be the class `Object`.
- 11 b) If  $C$  is not an instance of the class `Module`, raise a direct instance of the class `TypeError`.
- 12 c) Otherwise:
  - 13 1) Let  $N$  be the *constant-identifier*.
  - 14 2) If a binding with name  $N$  exists in the set of bindings of constants of  $C$ , the value of  
15 the *scoped-constant-reference* is the value of the binding.
  - 16 3) Otherwise:
    - 17 i) Let  $L$  be the included module list of  $C$ . Search each element of  $L$  in the reverse  
18 order for a binding of a constant with name  $N$ .
    - 19 ii) If the binding is found, the value of the *scoped-constant-reference* is the value of  
20 the binding.
    - 21 iii) Otherwise, if  $C$  is an instance of the class `Class`, search for a binding of a constant  
22 with name  $N$  from Step e) of 11.5.4.2.
    - 23 iv) Otherwise, create a direct instance of the class `Symbol` with name  $N$ , and let  $R$   
24 be that instance. Invoke the method `const_missing` on  $C$  with  $R$  as the only  
25 argument.

#### 26 **11.5.4.4 Global variables**

27 A *global-variable-identifier* is evaluated as follows:

- 28 a) Let  $N$  be the *global-variable-identifier*.
- 29 b) If a binding with name  $N$  exists in `[[global-variable-bindings]]`, the value of *global-variable-*  
30 *identifier* is the value of the binding.
- 31 c) Otherwise, the value of *global-variable-identifier* is **nil**.

#### 32 **11.5.4.5 Class variables**

33 A *class-variable-identifier* is evaluated as follows:

- 1 a) Let  $N$  be the *class-variable-identifier*. Let  $C$  be the first class or module in the list at the  
2 top of `[[class-module-list]]` which is not a singleton class.
- 3 b) Let  $CS$  be the set of classes which consists of  $C$  and all the superclasses of  $C$ . Let  $MS$  be  
4 the set of modules which consists of all the modules in the included module list of all classes  
5 in  $CS$ . Let  $CM$  be the union of  $CS$  and  $MS$ .
- 6 c) If a binding with name  $N$  exists in the set of bindings of class variables of only one of the  
7 classes or modules in  $CM$ , let  $V$  be the value of the binding.
- 8 d) If more than two classes or modules in  $CM$  have a binding with name  $N$  in the set of  
9 bindings of class variables, let  $V$  be the value of one of these bindings. Which binding is  
10 selected is implementation-defined.
- 11 e) If none of the classes or modules in  $CM$  has a binding with name  $N$  in the set of bindings  
12 of class variables, let  $S$  be a direct instance of the class `Symbol` with name  $N$  and raise a  
13 direct instance of the class `NameError` which has  $S$  as its name attribute.
- 14 f) The value of the *class-variable-identifier* is  $V$ .

#### 15 11.5.4.6 Instance variables

16 An *instance-variable-identifier* is evaluated as follows:

- 17 a) Let  $N$  be the *instance-variable-identifier*.
- 18 b) If a binding with name  $N$  exists in the set of bindings of instance variables of the current  
19 self, the value of the *instance-variable-identifier* is the value of the binding.
- 20 c) Otherwise, the value of the *instance-variable-identifier* is `nil`.

#### 21 11.5.4.7 Local variables or method invocations

##### 22 11.5.4.7.1 General description

23 An occurrence of a *local-variable-identifier* in a *variable-reference* is evaluated as either a refer-  
24 ence to a local variable or a method invocation.

##### 25 11.5.4.7.2 Determination of the type of local variable identifiers

26 Whether the occurrence of a *local-variable-identifier*  $I$  is a reference to a local variable or a  
27 method invocation is determined as follows:

- 28 a) Let  $P$  be the point of the program text where  $I$  occurs.
- 29 b) Let  $S$  be the innermost local variable scope which encloses  $P$  and which does not correspond  
30 to a *block* (see 9.2).
- 31 c) Let  $R$  be the region of the program text between the beginning of  $S$  and  $P$ .
- 32 d) If the same identifier as  $I$  occurs as a reference to a local variable in *variable-reference* in  
33  $R$ , then  $I$  is a reference to a local variable.

1 e) If the same identifier as  $I$  occurs in one of the the forms below in  $R$ , then  $I$  is a reference  
2 to a local variable.

3 • *mandatory-parameter*

4 • *optional-parameter-name*

5 • *array-parameter-name*

6 • *proc-parameter-name*

7 • *variable of left-hand-side*

8 • *variable of single-variable-assignment-expression*

9 • *variable of single-variable-assignment-statement*

10 • *variable of abbreviated-variable-assignment-expression*

11 • *variable of abbreviated-variable-assignment-statement*

12 f) Otherwise,  $I$  is a method invocation.

### 13 **11.5.4.7.3 Local variables**

14 If a *local-variable-identifier* is a reference to a local variable, it is evaluated as follows:

15 a) Let  $N$  be the *local-variable-identifier*.

16 b) Search for a binding of a local variable with name  $N$  as described in 9.2.

17 c) If a binding is found, the value of *local-variable-identifier* is the value of the binding.

18 d) Otherwise, the value of *local-variable-identifier* is **nil**.

### 19 **11.5.4.7.4 Method invocations**

20 If a *local-variable-identifier* is a method invocation, it is evaluated as follows:

21 a) Let  $N$  be the *local-variable-identifier*.

22 b) Create an empty list of arguments  $L$ , and invoke the method  $N$  on the current self with  $L$   
23 as the list of arguments (see 13.3.3).

### 24 **11.5.4.8 Pseudo variables**

#### 25 **11.5.4.8.1 General description**

#### 26 **Syntax**

---

1 *pseudo-variable* ::  
2     *nil-expression*  
3     | *true-expression*  
4     | *false-expression*  
5     | *self-expression*

---

6 NOTE A *pseudo-variable* has a similar form to a *local-variable-identifier*, but is not a variable.

#### 7 11.5.4.8.2 The nil expression

##### 8 Syntax

---

9 *nil-expression* ::  
10     **nil**

---

##### 11 Semantics

12 A *nil-expression* evaluates to **nil**, which is the only instance of the class NilClass (see 6.6).

#### 13 11.5.4.8.3 The true expression and the false expression

##### 14 Syntax

---

15 *true-expression* ::  
16     **true**

17 *false-expression* ::  
18     **false**

---

##### 19 Semantics

20 A *true-expression* evaluates to **true**, which is the only instance of the class TrueClass. A  
21 *false-expression* evaluates to **false**, which is the only instance of the class FalseClass (see 6.6).

#### 22 11.5.4.8.4 The self expression

##### 23 Syntax

---

24 *self-expression* ::  
25     **self**

---

## 1 Semantics

2 A *self-expression* evaluates to the value of the current self.

### 3 11.5.5 Object constructors

#### 4 11.5.5.1 Array constructor

##### 5 Syntax

---

6 *array-constructor* ::  
7 [ *indexing-argument-list*? ]

---

##### 8 Semantics

9 An *array-constructor* is evaluated as follows:

- 10 a) If there is an *indexing-argument-list*, construct a list of arguments from the *indexing-*  
11 *argument-list* as described in 11.3.2. Let  $L$  be the resulting list.
- 12 b) Otherwise, create an empty list of values  $L$ .
- 13 c) Create a direct instance of the class **Array** (see 15.2.12) which stores the values in  $L$  in the  
14 same order they are stored in  $L$ . Let  $O$  be the instance.
- 15 d) The value of the *array-constructor* is  $O$ .

#### 16 11.5.5.2 Hash constructor

##### 17 Syntax

---

18 *hash-constructor* ::  
19 { ( *association-list* [no *line-terminator* here] , ? )? }

20 *association-list* ::  
21 *association* ( [no *line-terminator* here] , *association* )\*

22 *association* ::  
23 *association-key* [no *line-terminator* here] => *association-value*

24 *association-key* ::  
25 *operator-expression*

26 *association-value* ::  
27 *operator-expression*

---

1 **Semantics**

2 a) A *hash-constructor* is evaluated as follows:

3 1) If there is an *association-list*, evaluate the *association-list*. The value of the *hash-*  
4 *constructor* is the resulting value.

5 2) Otherwise, create an empty direct instance of the class `Hash`. The value of the *hash-*  
6 *constructor* is the resulting instance.

7 b) An *association-list* is evaluated as follows:

8 1) Create an empty direct instance  $H$  of the class `Hash`.

9 2) For each *association*  $A_i$ , in the order it appears in the program text, take the following  
10 steps:

11 i) Evaluate the *operator-expression* of the *association-key* of  $A_i$ . Let  $K_i$  be the re-  
12 sulting value.

13 ii) Evaluate the *operator-expression* of the *association-value*. Let  $V_i$  be the resulting  
14 value.

15 iii) Store a pair of  $K_i$  and  $V_i$  in  $H$  by invoking the method `[]=` on  $H$  with  $K_i$  and  $V_i$   
16 as the arguments.

17 3) The value of the *association-list* is  $H$ .

18 **11.5.5.3 Range constructor**

19 **Syntax**

---

20 *range-constructor* ::  
21 *operator-OR-expression*  
22 | *operator-OR-expression*<sub>1</sub> [no *line-terminator* here] *range-operator* *operator-OR-*  
23 *expression*<sub>2</sub>

24 *range-operator* ::  
25 ..  
26 | ...

---

27 **Semantics**

28 A *range-constructor* of the form *operator-OR-expression*<sub>1</sub> *range-operator* *operator-OR-expression*<sub>2</sub>  
29 is evaluated as follows:

30 a) Evaluate the *operator-OR-expression*<sub>1</sub>. Let  $A$  be the resulting value.

31 b) Evaluate the *operator-OR-expression*<sub>2</sub>. Let  $B$  be the resulting value.

1 c) If the *range-operator* is the terminal “.”, construct a list *L* which contains three arguments:  
2 *A*, *B*, and **false**.

3 If the *range-operator* is the terminal “...”, construct a list *L* which contains three argu-  
4 ments: *A*, *B*, and **true**.

5 d) Invoke the method **new** on the class **Range** (see 15.2.14) with *L* as the list of arguments.  
6 The value of the *range-constructor* is the resulting value.

## 7 **12 Statements**

### 8 **12.1 General description**

#### 9 **Syntax**

---

10 *statement* ::  
11 *expression-statement*  
12 | *alias-statement*  
13 | *undef-statement*  
14 | *if-modifier-statement*  
15 | *unless-modifier-statement*  
16 | *while-modifier-statement*  
17 | *until-modifier-statement*  
18 | *rescue-modifier-statement*  
19 | *assignment-statement*

---

#### 20 **Semantics**

21 See 13.3.6 for *alias-statements*.

22 See 13.3.7 for *undef-statements*.

23 See 11.4.2 for *assignment-statements*.

### 24 **12.2 The expression statement**

#### 25 **Syntax**

---

26 *expression-statement* ::  
27 *expression*

---

#### 28 **Semantics**

29 An *expression-statement* is evaluated as follows:

30 a) Evaluate the *expression*.

1 b) The value of the *expression-statement* is the resulting value.

## 2 **12.3 The if modifier statement**

### 3 **Syntax**

---

4 *if-modifier-statement* ::  
5 *statement* [no *line-terminator* here] **if** *expression*

---

### 6 **Semantics**

7 An *if-modifier-statement* of the form *S if E*, where *S* is the *statement* and *E* is the *expression*,  
8 is evaluated as follows:

9 a) Evaluate the *if-expression* of the form **if E then S end**.

10 b) The value of the *if-modifier-statement* is the resulting value.

## 11 **12.4 The unless modifier statement**

### 12 **Syntax**

---

13 *unless-modifier-statement* ::  
14 *statement* [no *line-terminator* here] **unless** *expression*

---

### 15 **Semantics**

16 An *unless-modifier-statement* of the form *S unless E*, where *S* is the *statement* and *E* is the  
17 *expression*, is evaluated as follows:

18 a) Evaluate the *unless-expression* of the form **unless E then S end**.

19 b) The value of the *unless-modifier-statement* is the resulting value.

## 20 **12.5 The while modifier statement**

### 21 **Syntax**

---

22 *while-modifier-statement* ::  
23 *statement* [no *line-terminator* here] **while** *expression*

---

### 24 **Semantics**

25 A *while-modifier-statement* of the form *S while E*, where *S* is the *statement* and *E* is the  
26 *expression*, is evaluated as follows:

- 1 a) If  $S$  is a *begin-expression*, the behavior is implementation-defined.
- 2 b) Evaluate the *while-expression* of the form `while  $E$  do  $S$  end`.
- 3 c) The value of the *while-modifier-statement* is the resulting value.

## 4 12.6 The until modifier statement

### 5 Syntax

---

6 *until-modifier-statement* ::  
7 *statement* [no line-terminator here] `until expression`

---

### 8 Semantics

9 An *until-modifier-statement* of the form  $S$  `until  $E$` , where  $S$  is the *statement* and  $E$  is the  
10 *expression*, is evaluated as follows:

- 11 a) If  $S$  is a *begin-expression*, the behavior is implementation-defined.
- 12 b) Evaluate the *until-expression* of the form `until  $E$  do  $S$  end`.
- 13 c) The value of the *until-modifier-statement* is the resulting value.

## 14 12.7 The rescue modifier statement

### 15 Syntax

---

16 *rescue-modifier-statement* ::  
17 *main-statement-of-rescue-modifier-statement* [no line-terminator here]  
18 `rescue fallback-statement-of-rescue-modifier-statement`

19 *main-statement-of-rescue-modifier-statement* ::  
20 *statement*

21 *fallback-statement-of-rescue-modifier-statement* ::  
22 *statement* **but not** *statement-not-allowed-in-fallback-statement*

23 *statement-not-allowed-in-fallback-statement* ::  
24 *keyword-AND-expression*  
25 | *keyword-OR-expression*  
26 | *if-modifier-statement*  
27 | *unless-modifier-statement*  
28 | *while-modifier-statement*  
29 | *until-modifier-statement*  
30 | *rescue-modifier-statement*

---

## 1 Semantics

2 A *rescue-modifier-statement* is evaluated as follows:

- 3 a) Evaluate the *main-statement-of-rescue-modifier-statement*. Let  $V$  be the resulting value.
- 4 b) If a direct instance of the class `StandardError` is raised and not handled in Step a), evaluate  
5 *fallback-statement-of-rescue-modifier-statement*. The value of the *rescue-modifier-statement*  
6 is the resulting value.
- 7 c) If no instances of the class `Exception` are raised in Step a), or all the instances of the  
8 class `Exception` raised in Step a) are handled in Step a), the value of the *rescue-modifier-*  
9 *statement* is  $V$ .

## 10 13 Classes and modules

### 11 13.1 Modules

#### 12 13.1.1 General description

13 Every module is an instance of the class `Module` (see 15.2.2). However, not every instance of the  
14 class `Module` is a module because the class `Module` is a superclass of the class `Class`, an instance  
15 of which is not a module, but a class.

16 Modules have the following attributes:

17 **Included module list:** A list of modules included in the module. Module inclusion is  
18 described in 13.1.3.

19 **Constants:** A set of bindings of constants.

20 A binding of a constant is created by the following program constructs:

- 21 • *Assignments* (see 11.4.2)
- 22 • *Module-definitions* (see 13.1.2)
- 23 • *Class-definitions* (see 13.2.2)

24 **Class variables:** A set of bindings of class variables. A binding of a class variable is  
25 created by an *assignment* (see 11.4.2).

26 **Instance methods:** A set of method bindings. A method binding is created by a *method-*  
27 *definition* (see 13.3.1), a *singleton-method-definition* (see 13.4.3), an *alias-statement* (see  
28 13.3.6) or an *undef-statement* (see 13.3.7). The value of a method binding may be **undef**,  
29 which is the flag indicating that a method cannot be invoked (see 13.3.7).

#### 30 13.1.2 Module definition

### 31 Syntax

---

```

1  module-definition ::
2      module module-path module-body end

3  module-path ::
4      top-module-path
5      | module-name
6      | nested-module-path

7  module-name ::
8      constant-identifier

9  top-module-path ::
10     :: module-name

11 nested-module-path ::
12     primary-expression [no line-terminator here] :: module-name

13 module-body ::
14     body-statement

```

---

## 15 Semantics

16 A *module-definition* is evaluated as follows:

- 17 a) Determine the class or module  $C$  in which a binding with name *module-name* is to be  
18 created or modified as follows:
- 19 1) If the *module-path* is of the form *top-module-path*, let  $C$  be the class `Object`.
  - 20 2) If the *module-path* is of the form *module-name*, let  $C$  be the current class or module.
  - 21 3) If the *module-path* is of the form *nested-module-path*, evaluate the *primary-expression*.  
22 If the resulting value is an instance of the class `Module`, let  $C$  be the instance. Otherwise,  
23 raise a direct instance of the class `TypeError`.
- 24 b) Let  $N$  be the *module-name*.
- 25 1) If a binding with name  $N$  exists in the set of bindings of constants of  $C$ , let  $B$  be this  
26 binding. If the value of  $B$  is a module, let  $M$  be that module. Otherwise, raise a direct  
27 instance of the class `TypeError`.
  - 28 2) Otherwise, create a direct instance  $M$  of the class `Module`. Create a variable binding  
29 with name  $N$  and value  $M$  in the set of bindings of constants of  $C$ .
- 30 c) Modify the execution context as follows:

- 1) Create a new list which has the same members as that of the list at the top of `[[class-module-list]]`, and add  $M$  to the head of the newly created list. Push the list onto `[[class-module-list]]`.
  - 2) Push  $M$  onto `[[self]]`.
  - 3) Push the public visibility onto `[[default-method-visibility]]`.
  - 4) Push an empty set of bindings onto `[[local-variable-bindings]]`.
- d) Evaluate the *body-statement* (see 11.5.2.5) of the *module-body*. The value of the *module-definition* is the resulting value of the *body-statement*.
- e) Restore the execution context by removing the elements from the tops of `[[class-module-list]]`, `[[self]]`, `[[default-method-visibility]]`, and `[[local-variable-bindings]]`.

### 13.1.3 Module inclusion

Modules and classes can be extended by including other modules into them. When a module is included, the instance methods, the class variables, and the constants of the included module are available to the including class or module (see 11.5.4.5, 13.3.3, and 11.5.4.2).

Modules and classes can include other modules by invoking the method `include` (see 15.2.2.4.27) or the method `extend` (see 15.3.1.3.13).

A module  $M$  is included in another module  $N$  if and only if  $M$  is an element of the included module list of  $N$ . A module  $M$  is included in a class  $C$  if and only if  $M$  is an element of the included module list of  $C$ , or  $M$  is included in one of the superclasses of  $C$ .

## 13.2 Classes

### 13.2.1 General description

Every class is an instance of the class `Class` (see 15.2.3), which is a direct subclass of the class `Module`.

Classes have the same set of attributes as modules. In addition, every class has at most one single direct superclass.

### 13.2.2 Class definition

#### Syntax

---

```
class-definition ::
    class class-path [no line-terminator here] ( < superclass )? separator
    class-body end
```

```
class-path ::
    top-class-path
    | class-name
    | nested-class-path
```

```

1  class-name ::
2      constant-identifier

3  top-class-path ::
4      :: class-name

5  nested-class-path ::
6      primary-expression [no line-terminator here] :: class-name

7  superclass ::
8      expression

9  class-body ::
10     body-statement

```

---

## 11 Semantics

12 A *class-definition* is evaluated as follows:

- 13 a) Determine the class or module  $M$  in which the binding with name *class-name* is to be  
14 created or modified as follows:
- 15 1) If the *class-path* is of the form *top-class-path*, let  $M$  be the class `Object`.
  - 16 2) If the *class-path* is of the form *class-name*, let  $M$  be the current class or module.
  - 17 3) If the *class-path* is of the form *nested-class-path*, evaluate the *primary-expression*. If  
18 the resulting value is an instance of the class `Module`, let  $M$  be the instance. Otherwise,  
19 raise a direct instance of the class `TypeError`.
- 20 b) Let  $N$  be the *class-name*.
- 21 1) If a binding with name  $N$  exists in the set of bindings of constants of  $M$ , let  $B$  be that  
22 binding.
    - 23 i) If the value of  $B$  is an instance of the class `Class`, let  $C$  be the instance. Otherwise,  
24 raise a direct instance of the class `TypeError`.
    - 25 ii) If the *superclass* is present, evaluate it. If the resulting value does not correspond  
26 to the direct superclass of  $C$ , raise a direct instance of the class `TypeError`.
  - 27 2) Otherwise, create a direct instance of the class `Class`. Let  $C$  be that class.
    - 28 i) If the *superclass* is present, evaluate it. If the resulting value is not an instance  
29 of the class `Class`, raise a direct instance of the class `TypeError`. If the value of  
30 the *superclass* is a singleton class or the class `Class`, the behavior is unspecified.  
31 Otherwise, set the direct superclass of  $C$  to the value of the *superclass*.

- 1           ii) If the *superclass* of the *class-definition* is omitted, set the direct superclass of *C*
- 2           to the class `Object`.
  
- 3           iii) Create a singleton class, and associate it with *C*. It shall have the singleton class
- 4           of the direct superclass of *C* as one of its superclasses.
  
- 5           iv) Create a variable binding with name *N* and value *C* in the set of bindings of
- 6           constants of *M*.
  
- 7 c) Modify the execution context as follows:
  - 8       1) Create a new list which has the same members as that of the list at the top of `[[class-`
  - 9       `module-list]]`, and add *C* to the head of the newly created list. Push the list onto
  - 10       `[[class-module-list]]`.
  - 11       2) Push *C* onto `[[self]]`.
  - 12       3) Push the public visibility onto `[[default-method-visibility]]`.
  - 13       4) Push an empty set of bindings onto `[[local-variable-bindings]]`.
  
- 14 d) Evaluate the *body-statement* (see 11.5.2.5) of the *class-body*. The value of the *class-definition*
- 15       is the resulting value of the *body-statement*.
  
- 16 e) Restore the execution context by removing the elements from the tops of `[[class-module-list]]`,
- 17       `[[self]]`, `[[default-method-visibility]]`, and `[[local-variable-bindings]]`.

### 18 **13.2.3 Inheritance**

19 A class inherits attributes of its superclasses. Inheritance means that a class implicitly contains  
20 all attributes of its superclasses, as described below:

- 21 • Constants and class variables of superclasses can be referred to (see 11.5.4.2 and 11.5.4.5).
- 22 • Singleton methods of superclasses can be invoked (see 13.4).
- 23 • Instance methods defined in superclasses can be invoked on an instance of their subclasses
- 24     (see 13.3.3).

### 25 **13.2.4 Instance creation**

26 A direct instance of a class can be created by invoking the method `new` (see 15.2.3.3.3) on the  
27 class.

## 28 **13.3 Methods**

### 29 **13.3.1 Method definition**

#### 30 **Syntax**

---

```

1  method-definition ::
2      def defined-method-name [no line-terminator here] method-parameter-part
3          method-body end

4  defined-method-name ::
5      method-name
6      | assignment-like-method-identifier

7  method-body ::
8      body-statement

```

---

9 The following constructs shall not be present in the *method-parameter-part* or the *method-body*:

- 10 • A *class-definition*.
- 11 • A *module-definition*.
- 12 • A *single-variable-assignment*, where its *variable* is a *constant-identifier*.
- 13 • A *scoped-constant-assignment*.
- 14 • A *multiple-assignment-statement* in which there exists a *left-hand-side* of any of the follow-  
15 ing forms:
  - 16 — *constant-identifier*;
  - 17 — *primary-expression* [no *line-terminator* here] (*.* | *::*) (*local-variable-identifier* | *constant-*  
18 *identifier*);
  - 19 — *:: constant-identifier*.

20 However, those constructs may occur within a *singleton-class-definition* in the *method-parameter-*  
21 *part* or the *method-body*.

## 22 Semantics

23 A method is defined by a *method-definition* or a *singleton-method-definition* (see 13.4.3), and has  
24 the *method-parameter-part* and the *method-body* of the *method-definition* or *singleton-method-*  
25 *definition*. The *method-body* is evaluated when the method is invoked (see 13.3.3). The evalu-  
26 ation of the *method-body* is the evaluation of its *body-statement* (see 11.5.2.5). In addition, a  
27 method has the following attributes:

28 **Class module list:** The list of classes and modules which is the top element of `[[class-`  
29 `module-list]]` when the method is defined.

30 **Defined name:** The name with which the method is defined.

31 **Visibility:** The visibility of the method (see 13.3.5).

1 A class or a module can define a new method with the same name as the name of a method in  
2 one of its superclasses or included modules of the class or module. In that case, the new method  
3 is said to **override** the method in the superclass or the included module.

4 A *method-definition* is evaluated as follows:

5 a) Let  $N$  be the *defined-method-name*.

6 b) Create a method  $U$  defined by the *method-definition*. Initialize the attributes of  $U$  as  
7 follows:

8 • The class module list is the element at the top of `[[class-module-list]]`.

9 • The defined name is  $N$ .

10 • The visibility is:

11 — If the current class or module is a singleton class, then the current visibility.

12 — Otherwise, if  $N$  is `initialize` or `initialize_copy`, then the private visibility.

13 — Otherwise, the current visibility.

14 c) If a method binding with name  $N$  exists in the set of bindings of instance methods of the  
15 current class or module, let  $V$  be the value of that binding.

16 1) If  $V$  is `undef`, the evaluation of the *method-definition* is implementation-defined.

17 2) Replace the value of the binding with  $U$ .

18 d) Otherwise, create a method binding with name  $N$  and value  $U$  in the set of bindings of  
19 instance methods of the current class or module.

20 e) The value of the *method-definition* is implementation-defined.

### 21 13.3.2 Method parameters

#### 22 Syntax

---

23 *method-parameter-part* ::  
24 ( *parameter-list*<sup>?</sup> )  
25 | *parameter-list*<sup>?</sup> *separator*

26 *parameter-list* ::  
27 *mandatory-parameter-list* ( , *optional-parameter-list* )<sup>?</sup>  
28 ( , *array-parameter* )<sup>?</sup> ( , *proc-parameter* )<sup>?</sup>  
29 | *optional-parameter-list* ( , *array-parameter* )<sup>?</sup> ( , *proc-parameter* )<sup>?</sup>  
30 | *array-parameter* ( , *proc-parameter* )<sup>?</sup>  
31 | *proc-parameter*

```

1  mandatory-parameter-list ::
2      mandatory-parameter
3      | mandatory-parameter-list , mandatory-parameter

4  mandatory-parameter ::
5      local-variable-identifier

6  optional-parameter-list ::
7      optional-parameter
8      | optional-parameter-list , optional-parameter

9  optional-parameter ::
10     optional-parameter-name = default-parameter-expression

11  optional-parameter-name ::
12     local-variable-identifier

13  default-parameter-expression ::
14     operator-expression

15  array-parameter ::
16     * array-parameter-name
17     | *

18  array-parameter-name ::
19     local-variable-identifier

20  proc-parameter ::
21     & proc-parameter-name

22  proc-parameter-name ::
23     local-variable-identifier

```

---

24 All the *local-variable-identifiers* of *mandatory-parameters*, *optional-parameter-names*, the *array-*  
25 *parameter-name*, and the *proc-parameter-name* in a *parameter-list* shall be different.

## 26 Semantics

27 There are four kinds of parameters as described below. How those parameters are bound to the  
28 actual arguments is described in 13.3.3.

29 **Mandatory parameters:** These parameters are represented by *mandatory-parameters*.  
30 For each mandatory parameter, a corresponding actual argument shall be given when the  
31 method is invoked.

32 **Optional parameters:** These parameters are represented by *optional-parameters*. Each

1 optional parameter consists of a parameter name represented by *optional-parameter-name*  
2 and an expression represented by *default-parameter-expression*. For each optional parame-  
3 ter, when there is no corresponding argument in the list of arguments given to the method  
4 invocation, the value of the *default-parameter-expression* is used as the value of the argu-  
5 ment.

6 **An array parameter:** This parameter is represented by *array-parameter-name*. Let  $N$  be  
7 the number of arguments, excluding a *block-argument*, given to a method invocation. If  $N$   
8 is more than the sum of the number of mandatory parameters and optional parameters, this  
9 parameter is bound to a direct instance of the class `Array` containing the extra arguments  
10 excluding a *block-argument*. Otherwise, the parameter is bound to an empty direct instance  
11 of the class `Array`. If an *array-parameter* is of the form “\*”, those extra arguments are  
12 ignored.

13 **A proc parameter:** This parameter is represented by *proc-parameter-name*. The param-  
14 eter is bound to a direct instance of the class `Proc` which represents the block passed to the  
15 method invocation.

### 16 13.3.3 Method invocation

17 The way in which a list of arguments is created is described in 11.3.

18 Given the receiver  $R$ , the method name  $M$ , and the list of arguments  $A$ , take the following steps:

- 19 a) If the method is invoked with a block, let  $B$  be the block. Otherwise, let  $B$  be block-not-  
20 given.
- 21 b) Let  $C$  be the singleton class of  $R$  if  $R$  has a singleton class. Otherwise, let  $C$  be the class  
22 of  $R$ .
- 23 c) Search for a method binding with name  $M$ , starting from  $C$  as described in 13.3.4.
- 24 d) If a binding is found and its value is not `undef`, let  $V$  be the value of the binding.
- 25 e) Otherwise, if  $M$  is `method_missing`, the behavior is unspecified. If  $M$  is not `method_missing`,  
26 add a direct instance of the class `Symbol` with name  $M$  to the head of  $A$ , and invoke the  
27 method `method_missing` (see 15.3.1.3.30) on  $R$  with  $A$  as arguments and  $B$  as the block.  
28 Let  $O$  be the resulting value, and go to Step j).
- 29 f) Check the visibility of  $V$  to see whether the method can be invoked (see 13.3.5). If the  
30 method cannot be invoked, add a direct instance of the class `Symbol` with name  $M$  to the  
31 head of  $A$ , and invoke the method `method_missing` on  $R$  with  $A$  as arguments and  $B$  as  
32 the block. Let  $O$  be the resulting value, and go to Step j).
- 33 g) Modify the execution context as follows:
  - 34 1) Push the class module list of  $V$  onto `[[class-module-list]]`.
  - 35 2) Push  $R$  onto `[[self]]`.
  - 36 3) Push  $M$  onto `[[invoked-method-name]]`.
  - 37 4) Push the public visibility to `[[default-method-visibility]]`.

- 1       5) Push the defined name of  $V$  onto  $\llbracket\text{defined-method-name}\rrbracket$ .
- 2       6) Push  $B$  onto  $\llbracket\text{block}\rrbracket$ .
- 3       7) Push an empty set of local variable bindings onto  $\llbracket\text{local-variable-bindings}\rrbracket$ .
- 4 h) Evaluate the *method-parameter-part* of  $V$  as follows:
  - 5       1) Let  $L$  be the *parameter-list* of the *method-parameter-part*.
  - 6       2) Let  $P_m$ ,  $P_o$ , and  $P_a$  be the *mandatory-parameters* of the *mandatory-parameter-list*,  
7       the *optional-parameters* of the *optional-parameter-list*, and the *array-parameter* of  $L$ ,  
8       respectively. Let  $N_A$ ,  $N_{P_m}$ , and  $N_{P_o}$  be the number of elements of  $A$ ,  $P_m$ , and  $P_o$   
9       respectively. If there are no *mandatory-parameters* or *optional-parameters*, let  $N_{P_m}$   
10       and  $N_{P_o}$  be 0. Let  $S_b$  be the current set of local variable bindings.
  - 11       3) If  $N_A$  is smaller than  $N_{P_m}$ , raise a direct instance of the class **ArgumentError**.
  - 12       4) If the method does not have  $P_a$  and  $N_A$  is larger than the sum of  $N_{P_m}$  and  $N_{P_o}$ , raise  
13       a direct instance of the class **ArgumentError**.
  - 14       5) Otherwise, for each  $i$ th argument  $A_i$  in  $A$ , in the same order in  $A$ , take the following  
15       steps:
    - 16       i) Let  $P_i$  be the  $i$ th *mandatory-parameter* or *optional-parameter* in the order it ap-  
17       pears in  $L$ .
    - 18       ii) If such  $P_i$  does not exist, go to Step h) 6).
    - 19       iii) If  $P_i$  is a mandatory parameter, let  $n$  be the *mandatory-parameter*. If  $P_i$  is an op-  
20       tional parameter, let  $n$  be the *optional-parameter-name*. Create a variable binding  
21       with name  $n$  and value  $A_i$  in  $S_b$ .
  - 22       6) If  $N_A$  is larger than the sum of  $N_{P_m}$  and  $N_{P_o}$ , and  $P_a$  exists:
    - 23       i) Create a direct instance  $X$  of the class **Array** whose length is the number of extra  
24       arguments.
    - 25       ii) Store each extra arguments into  $X$ , preserving the order in which they occur in  
26       the list of arguments.
    - 27       iii) Let  $n$  be the *array-parameter-name* of  $P_a$ .
    - 28       iv) Create a variable binding with name  $n$  and value  $X$  in  $S_b$ .
  - 29       7) If  $N_A$  is smaller than the sum of  $N_{P_m}$  and  $N_{P_o}$ :
    - 30       i) For each optional parameter  $P_{O_i}$  to which no argument corresponds, evaluate the  
31       *default-parameter-expression* of  $P_{O_i}$ , and let  $X$  be the resulting value.
    - 32       ii) Let  $n$  be the *optional-parameter-name* of  $P_{O_i}$ .

- 1           iii) Create a variable binding with name  $n$  and value  $X$  in  $S_b$ .
- 2       8) If  $N_A$  is smaller than or equal to the sum of  $N_{P_m}$  and  $N_{P_o}$ , and  $P_a$  exists:
  - 3           i) Create an empty direct instance  $X$  of the class **Array**.
  - 4           ii) Let  $n$  be the *array-parameter-name* of  $P_a$ .
  - 5           iii) Create a variable binding with name  $n$  and value  $X$  in  $S_b$ .
- 6       9) If the *proc-parameter* of  $L$  is present, let  $D$  be the top of `[[block]]`.
  - 7           i) If  $D$  is block-not-given, let  $X$  be **nil**.
  - 8           ii) Otherwise, invoke the method **new** on the class **Proc** with an empty list of arguments and  $D$  as the block. Let  $X$  be the resulting value of the method invocation.
  - 9           iii) Let  $n$  be the *proc-parameter-name* of *proc-parameter*.
  - 10          iii) Let  $n$  be the *proc-parameter-name* of *proc-parameter*.
  - 11          iv) Create a variable binding with name  $n$  and value  $X$  in  $S_b$ .
- 12   i) Evaluate the *method-body* of  $V$ .
  - 13          1) If the evaluation of the *method-body* is terminated by a *return-expression*:
    - 14           i) If the *jump-argument* of the *return-expression* is present, let  $O$  be the value of the
    - 15           *jump-argument*.
    - 16           ii) Otherwise, let  $O$  be **nil**.
  - 17          2) Otherwise, let  $O$  be the resulting value of the evaluation.
- 18   j) Restore the execution context by removing the elements from the tops of `[[class-module-`  
19       `list]]`, `[[self]]`, `[[invoked-method-name]]`, `[[default-method-visibility]]`, `[[defined-method-name]]`,  
20       `[[block]]`, and `[[local-variable-bindings]]`.
- 21   k) The value of the method invocation is  $O$ .

22 The method invocation or the *super-expression* [see 11.3.4 d)] which corresponds to the set of  
23 items on the tops of all the attributes of the execution context modified in Step g), except  
24 `[[local-variable-bindings]]`, is called the **current method invocation**.

### 25 13.3.4 Method lookup

26 Method lookup is the process by which a binding of an instance method is resolved.

27 Given a method name  $M$  and a class or a module  $C$  which is initially searched for the binding  
28 of the method, the method binding is resolved as follows:

- 29 a) If a method binding with name  $M$  exists in the set of bindings of instance methods of  $C$ ,
- 30 let  $B$  be that binding.

1 b) Otherwise, let  $L_m$  be the included module list of  $C$ . Search for a method binding with name  
2  $M$  in the set of bindings of instance methods of each module in  $L_m$ . Examine modules in  
3  $L_m$  in reverse order.

4 1) If a binding is found, let  $B$  be that binding.

5 2) Otherwise:

6 i) If  $C$  does not have a direct superclass, the binding is considered not resolved.

7 ii) Otherwise, let new  $C$  be the direct superclass of  $C$ , and continue processing from  
8 Step a).

9  $B$  is the resolved method binding.

10

### 11 **13.3.5 Method visibility**

#### 12 **13.3.5.1 General description**

13 Methods are categorized into one of public, private, or protected methods according to the  
14 conditions under which the method invocation is allowed. The attribute of a method which  
15 determines these conditions is called the **visibility** of the method.

#### 16 **13.3.5.2 Public methods**

17 A public method is a method whose visibility attribute is set to the public visibility.

18 A public method can be invoked on an object anywhere within a program.

#### 19 **13.3.5.3 Private methods**

20 A private method is a method whose visibility attribute is set to the private visibility.

21 A private method cannot be invoked with an explicit receiver, i.e., a private method cannot  
22 be invoked if a *primary-expression* or a *chained-method-invocation* occurs at the position which  
23 corresponds to the method receiver in the method invocation, except for a method invocation  
24 of any of the following forms where the *primary-expression* is a *self-expression*.

25 ●) *single-method-assignment*

26 ● *abbreviated-method-assignment*

27 ● *single-indexing-assignment*

28 ● *abbreviated-indexing-assignment*

#### 29 **13.3.5.4 Protected methods**

30 A protected method is a method whose visibility attribute is set to the protected visibility.

31 A protected method can be invoked if and only if the following condition holds:

- 1 • Let  $M$  be an instance of the class `Module` in which the binding of the method exists.
- 2  $M$  is included in the current self, or  $M$  is the class of the current self or one of its superclasses.
- 3 If  $M$  is a singleton class, whether the method can be invoked or not may be determined in a
- 4 implementation-defined way.

### 5 **13.3.5.5 Visibility change**

6 The visibility of methods can be changed with the methods `public` (see 15.2.2.4.38), `private`  
7 (see 15.2.2.4.36), and `protected` (see 15.2.2.4.37), which are defined in the class `Module`.

### 8 **13.3.6 The alias statement**

#### 9 **Syntax**

---

10 *alias-statement* ::  
11     **alias** *new-name* *aliased-name*

12 *new-name* ::  
13     *defined-method-name*  
14     | *symbol*

15 *aliased-name* ::  
16     *defined-method-name*  
17     | *symbol*

---

#### 18 **Semantics**

19 An *alias-statement* is evaluated as follows:

20 a) Evaluate the *new-name* as follows:

- 21 1) If the *new-name* is of the form *defined-method-name*, let  $N$  be the *defined-method-name*.
- 22 2) If the *new-name* is of the form *symbol*, evaluate it. Let  $N$  be the name of the resulting
- 23 instance of the class `Symbol`.

24 b) Evaluate the *aliased-name* as follows:

- 25 1) If *aliased-name* is of the form *defined-method-name*, let  $A$  be the *defined-method-name*.
- 26 2) If *aliased-name* is of the form *symbol*, evaluate it. Let  $A$  be the name of the resulting
- 27 instance of the class `Symbol`.

28 c) Let  $C$  be the current class or module.

29 d) Search for a method binding with name  $A$ , starting from  $C$  as described in 13.3.4.

- 1 e) If a binding is found and its value is not `undef`, let  $V$  be the value of the binding.
- 2 f) Otherwise, let  $S$  be a direct instance of the class `Symbol` with name  $A$  and raise a direct  
3 instance of the class `NameError` which has  $S$  as its name attribute.
- 4 g) If a method binding with name  $N$  exists in the set of bindings of instance methods of the  
5 current class or module, replace the value of the binding with  $V$ .
- 6 h) Otherwise, create a method binding with name  $N$  and value  $V$  in the set of bindings of  
7 instance methods of the current class or module.
- 8 i) The value of the *alias-statement* is **nil**.

### 9 13.3.7 The undef statement

#### 10 Syntax

---

11 *undef-statement* ::  
12     **undef** *undef-list*

13 *undef-list* ::  
14     *method-name-or-symbol* ( , *method-name-or-symbol* )\*

15 *method-name-or-symbol* ::  
16     *defined-method-name*  
17     | *symbol*

---

#### 18 Semantics

19 An *undef-statement* is evaluated as follows:

- 20 a) For each *method-name-or-symbol* of the *undef-list*, take the following steps:
  - 21 1) Let  $C$  be the current class or module.
  - 22 2) If the *method-name-or-symbol* is of the form *defined-method-name*, let  $N$  be the *defined-*  
23 *method-name*. Otherwise, evaluate the *symbol*. Let  $N$  be the name of the resulting  
24 instance of the class `Symbol`.
  - 25 3) Search for a method binding with name  $N$ , starting from  $C$  as described in 13.3.4.
  - 26 4) If a binding is found and its value is not `undef`:
    - 27 i) If the binding is found in  $C$ , replace the value of the binding with `undef`.
    - 28 ii) Otherwise, create a method binding with name  $N$  and value `undef` in the set of  
29 bindings of instance methods of  $C$ .
  - 30 5) Otherwise, let  $S$  be a direct instance of the class `Symbol` with name  $N$  and raise a  
31 direct instance of the class `NameError` which has  $S$  as its name attribute.

1 b) The value of the *undef-statement* is **nil**.

## 2 **13.4 Singleton classes**

### 3 **13.4.1 General description**

4 A singleton class is an object which is associated with another object. A singleton class modifies  
5 the behavior of an object when associated with it. When such an association is made, the  
6 singleton class is called the singleton class of the object, and the object is called the primary  
7 associated object of the singleton class.

8 An object has at most one singleton class. When an object is created, it shall not be associated  
9 with any singleton classes unless the object is an instance of the class **Class**. Singleton classes  
10 are associated with an object by evaluation of a program construct such as a *singleton-method-*  
11 *definition* or a *singleton-class-definition*. However, when an instance of the class **Class** is created,  
12 it shall already have been associated with its singleton class.

13 Normally, a singleton class shall be associated with only its primary associated object; however,  
14 the singleton class of an instance of the class **Class** may be associated with some additional  
15 instances of the class **Class** which are not the primary associated objects of any other singleton  
16 classes, in an implementation-defined way. Once associated, the primary associated object of  
17 a singleton class shall not be dissociated from its singleton class; however the aforementioned  
18 additional associated instances of the class **Class** are dissociated from their singleton class when  
19 they become the primary associated object of another singleton class [see 13.4.2 e) and 13.4.3  
20 e)].

21 Every singleton class is an instance of the class **Class** (see 15.2.3), and has the same set of  
22 attributes as classes.

23 The direct superclass of a singleton class is implementation-defined. However, a singleton class  
24 shall be a subclass of the class of the object with which it is associated.

25 NOTE 1 For example, the singleton class of the class **Object** is a subclass of the class **Class** because  
26 the class **Object** is a direct instance of the class **Class**. Therefore, the instance methods of the class  
27 **Class** can be invoked on the class **Object**.

28 The singleton class of a class which has a direct superclass shall satisfy the following condition:

- 29 • Let  $E_c$  be the singleton class of a class  $C$ , and let  $S$  be the direct superclass of  $C$ , and let  
30  $E_s$  be the singleton class of  $S$ . Then,  $E_c$  have  $E_s$  as one of its superclasses.

31 NOTE 2 This requirement enables classes to inherit singleton methods from its superclasses. For exam-  
32 ple, the singleton class of the class **File** has the singleton class of the class **IO** as its superclass. Thereby,  
33 the class **File** inherits the singleton method **open** of the class **IO**.

34 Although singleton classes are instances of the class **Class**, they cannot create an instance of  
35 themselves. When the method **new** is invoked on a singleton class, a direct instance of the class  
36 **TypeError** shall be raised [see 15.2.3.3.3 a)].

37 Whether a singleton class can be a superclass of other classes is unspecified [see 13.2.2 b) 2) i)  
38 and 15.2.3.3.1 c)].

39 Whether a singleton class can have class variables or not is implementation-defined.

## 1 13.4.2 Singleton class definition

### 2 Syntax

---

3 *singleton-class-definition* ::  
4 `class << expression separator singleton-class-body end`

5 *singleton-class-body* ::  
6 *body-statement*

---

### 7 Semantics

8 A *singleton-class-definition* is evaluated as follows:

- 9 a) Evaluate the *expression*. Let  $O$  be the resulting value. If  $O$  is an instance of the class  
10 `Integer` or the class `Symbol`, a direct instance of the class `TypeError` may be raised.
- 11 b) If  $O$  is one of `nil`, `true`, or `false`, let  $E$  be the class of  $O$  and go to Step f).
- 12 c) If  $O$  is not associated with a singleton class, create a new singleton class. Let  $E$  be the  
13 newly created singleton class, and associate  $O$  with  $E$  as its primary associated object.
- 14 d) If  $O$  is associated with a singleton class as its primary associated object, let  $E$  be that  
15 singleton class.
- 16 e) If  $O$  is associated with a singleton class not as its primary associated object, dissociate  
17  $O$  from the singleton class, and create a new singleton class. Let  $E$  be the newly created  
18 singleton class, and associate  $O$  with  $E$  as its primary associated object.
- 19 f) Modify the execution context as follows:
- 20 1) Create a new list which consists of the same elements as the list at the top of `[[class-`  
21 `module-list]]` and add  $E$  to the head of the newly created list. Push the list onto  
22 `[[class-module-list]]`.
- 23 2) Push  $E$  onto `[[self]]`.
- 24 3) Push the public visibility onto `[[default-method-visibility]]`.
- 25 4) Push an empty set of bindings onto `[[local-variable-bindings]]`.
- 26 g) Evaluate the *singleton-class-body*. The value of the *singleton-class-definition* is the value of  
27 the *singleton-class-body*.
- 28 h) Restore the execution context by removing the elements from the tops of `[[class-module-list]]`,  
29 `[[self]]`, `[[default-method-visibility]]`, and `[[local-variable-bindings]]`.

## 30 13.4.3 Singleton method definition

### 31 Syntax

---

```

1  singleton-method-definition ::
2      def singleton ( . | :: ) defined-method-name [no line-terminator here]
3          method-parameter-part method-body end

4  singleton ::
5      variable-reference
6      | ( expression )

```

---

## 7 Semantics

8 A *singleton-method-definition* is evaluated as follows:

- 9 a) Evaluate the *singleton*. Let  $S$  be the resulting value. If  $S$  is an instance of the class `Integer` or the class `Symbol`, a direct instance of the class `TypeError` may be raised.
- 10
- 11 b) If  $S$  is one of **nil**, **true**, or **false**, let  $E$  be the class of  $O$  and go to Step f).
- 12 c) If  $S$  is not associated with a singleton class, create a new singleton class. Let  $E$  be the newly created singleton class, and associate  $S$  with  $E$  as its primary associated object.
- 13
- 14 d) If  $S$  is associated with a singleton class as its primary associated object, let  $E$  be that singleton class.
- 15
- 16 e) If  $S$  is associated with a singleton class not as its primary associated object, dissociate  $S$  from the singleton class, and create a new singleton class. Let  $E$  be the newly created singleton class, and associate  $S$  with  $E$  as its primary associated object.
- 17
- 18
- 19 f) Let  $N$  be the *defined-method-name*.
- 20 g) Create a method  $U$  defined by the *singleton-method-definition*.  $U$  has the *method-parameter-part* and the *method-body* of the *singleton-method-definition* as described in 13.3.1. Initialize the attributes of  $U$  as follows:
  - 21 • The class module list is the element at the top of `[[class-module-list]]`.
  - 22
  - 23 • The defined name is  $N$ .
  - 24
  - 25 • The visibility is the public visibility.
- 26 h) If a method binding with name  $N$  exists in the set of bindings of instance methods of  $E$ , let  $V$  be the value of that binding.
  - 27
  - 28 1) If  $V$  is undef, the evaluation of the *singleton-method-definition* is implementation-defined.
  - 29
  - 30 2) Replace the value of the binding with  $U$ .
- 31 i) Otherwise, create a method binding with name  $N$  and value  $U$  in the set of bindings of instance methods of  $E$ .
- 32

1 j) The value of the *singleton-method-definition* is implementation-defined.

## 2 14 Exceptions

### 3 14.1 General description

4 If an instance of the class `Exception` is raised, the current evaluation process stops, and control  
5 is transferred to a program construct that can handle this exception.

### 6 14.2 Cause of exceptions

7 An exception is raised when:

- 8 • the method `raise` (see 15.3.1.2.12) is invoked.
- 9 • an exceptional condition occurs as described in various parts of this document.

10 Only instances of the class `Exception` shall be raised.

### 11 14.3 Exception handling

12 Exceptions are handled by a *body-statement*, an *assignment-with-rescue-modifier*, or a *rescue-*  
13 *modifier-statement*. These program constructs are called **exception handlers**. When an ex-  
14 ception handler is handling an exception, the exception being handled is called the **current**  
15 **exception**.

16 When an exception is raised, it is handled by an exception handler. This exception handler is  
17 determined as follows:

18 a) Let  $S$  be the innermost local variable scope which lexically encloses the location where  
19 the exception is raised, and which corresponds to one of a *program*, a *method-definition*, a  
20 *singleton-method-definition*, or a *block*.

21 b) Test each exception handler in  $S$  which lexically encloses the location where the exception  
22 is raised from the innermost to the outermost.

23 • An *assignment-with-rescue-modifier* is considered to handle the exception if the excep-  
24 tion is an instance of the class `StandardError` (see 11.4.2.5), except when the exception  
25 is raised in its *operator-expression*<sub>2</sub>. In this case, *assignment-with-rescue-modifier* does  
26 not handle the exception.

27 • A *rescue-modifier-statement* is considered to handle the exception if the exception is  
28 an instance of the class `StandardError` (see 12.7), except when the exception is raised  
29 in its *fallback-statement-of-rescue-modifier-statement*. In this case, *rescue-modifier-*  
30 *statement* does not handle the exception.

31 • A *body-statement* is considered to handle the exception if one of its *rescue-clauses* is  
32 considered to handle the exception (see 11.5.2.5), except when the exception is raised  
33 in one of its *rescue-clauses*, *else-clause*, or *ensure-clause*. In this case, *body-statement*  
34 does not handle the exception. If an *ensure-clause* of a *body-statement* is present, it is  
35 evaluated even if the handler does not handle the exception (see 11.5.2.5).

- 1 c) If an exception handler which can handle the exception is found in *S*, terminate the search  
2 for the exception handler. Continue evaluating the program as defined for the relevant  
3 construct (see 11.4.2.5, 11.5.2.5, and 12.7).
- 4 d) If none of the exception handlers in *S* can handle the exception:
- 5 1) If *S* corresponds to a *method-definition* or a *singleton-method-definition*, terminate Step  
6 h) or Step i) of 13.3.3, and take Step j) of the current method invocation (see 13.3.3).  
7 Continue the search from Step a), under the assumption that the exception is raised  
8 at the location where the method is invoked.
- 9 2) If *S* corresponds to a *block*, terminate the evaluation of the current *block*, and take  
10 Step f) of 11.3.3. Continue the search from Step a), under the assumption that the  
11 exception is raised at the location where the block is called.
- 12 3) If *S* corresponds to a *program*, terminate the evaluation of the *program*, take Step d)  
13 of 10.1, and print the information of the exception in an implementation-defined way.

## 14 15 Built-in classes and modules

### 15 15.1 General description

16 Built-in classes and modules are classes and modules which are already created before execution  
17 of a program (see 7.2).

18 Built-in classes and modules are respectively specified in 15.2 and 15.3. A built-in class or  
19 module is specified by describing the following attributes (see 13.1.1 and 13.2.1):

- 20 • The direct superclass (for built-in classes only).
- 21 • The include module list.
- 22 • Constants.
- 23 • Singleton methods, i.e. instance methods of the singleton class of the built-in class or  
24 module. The class module list of a singleton method of the built-in class or module consists  
25 of two elements: the first is the singleton class of the built-in class or module; the second is  
26 the class `Object`.
- 27 • Instance methods. The class module list (see 13.3.1) of an instance method of the built-in  
28 class or module consists of two elements: the first is the built-in class or module; the second  
29 is the class `Object`.

30 The set of bindings of class variables of a built-in class or module is an empty set.

31 NOTE A built-in class or module is not created by a *class-definition* or *module-definition* in a program  
32 text, but is created as a class or module whose attributes are described in 15.2 or 15.3 in advance prior  
33 to an execution of a program.

34 A conforming processor may provide the following additional attributes and/or values.

- 1 • A specific initial value for an attribute defined in this document whose initial value is not  
2 specified in this document;
- 3 • Constants, singleton methods, instance methods;
- 4 • Additional optional parameters or array parameters for methods specified in this document;
- 5 • Additional inclusion of modules into built-in classes/modules.

6 In 15.2 and 15.3, the following notations are used:

- 7 • Each subclause of 15.2 and 15.3 (e.g., 15.2.1) specifies a built-in class or module. The title  
8 of the subclause is the name of the built-in class or module. The name is used as the name  
9 of a constant binding in the class `Object` (see 15.2.1.4).
- 10 • A built-in class except the class `Object` (see 15.2.1) has, as its direct superclass, the class  
11 described in the subclause titled “Direct superclass” in the subclause specifying the built-in  
12 class.
- 13 • When a subclause specifying a built-in class or module contains a subclause titled “Included  
14 modules”, the built-in class or module includes (see 13.1.3) the modules listed in that  
15 subclause in the order of that listing.
- 16 • Each subclause in a subclause titled “Singleton methods” with a title of the form *C.m*  
17 specifies the singleton method *m* of the class *C*.
- 18 • Each subclause in a subclause titled “Instance methods” with a title of the form *C#m*  
19 specifies the instance method *m* of the class *C*.
- 20 • The parameter specification of a method is described in the form of *method-parameter-part*  
21 (see 13.3.2).

22 EXAMPLE 1 The following example defines the parameter specification of a method `sample`.

---

23 `sample( arg1, arg2, opt=expr, *ary, &blk )`

---

- 24 • A singleton method name is prefixed by the name of the class or the module, and a dot (`.`).

25 EXAMPLE 2 The following example defines the parameter specification of a singleton method  
26 `sample` of a class `SampleClass`:

---

27 `SampleClass.sample( arg1, arg2, opt=expr, *ary, &blk )`

---

- 28 • Next to the parameter specification, the visibility and the behavior of the method are  
29 specified.

30 The visibility, which is any one of `public`, `protected`, or `private`, is specified after the label  
31 named “Visibility:”.

32 The behavior, which is the steps which shall be taken while evaluating the *method-body* of  
33 the method [see 13.3.3 i)], is specified after the label named “Behavior:”.

1 In these steps, a reference to the name of an argument in the parameter specification is  
2 considered to be the object bound to the local variables of the same name.

3 • The phrase “call *block* with *X* as the argument” indicates that the block corresponding to  
4 the proc parameter *block* is called as described in 11.3.3 with *X* as the argument to the  
5 block call.

6 • The phrase “return *X*” indicates that the evaluation of the *method-body* is terminated at  
7 that point, and *X* is the value of the *method-body*.

8 • The phrase “the name designated by *N*” means the result of the following steps:

9 a) If *N* is an instance of the class `Symbol`, the name of *N*.

10 b) If *N* is an instance of the class `String`, the content (see 15.2.10.1) of *N*.

11 c) Otherwise, the behavior of the method is unspecified.

## 12 **15.2 Built-in classes**

### 13 **15.2.1 Object**

#### 14 **15.2.1.1 General description**

15 The class `Object` is an implicit direct superclass for other classes. That is, if the direct superclass  
16 of a class is not specified explicitly in the class definition, the direct superclass of the class is  
17 the class `Object` (see 13.2.2).

18 All built-in classes and modules can be referred to through constants of the class `Object` (see  
19 15.2.1.4).

#### 20 **15.2.1.2 Direct superclass**

21 The class `Object` does not have a direct superclass, or may have an implementation-defined  
22 superclass.

#### 23 **15.2.1.3 Included modules**

24 The following module is included in the class `Object`.

- 25 • `Kernel`

#### 26 **15.2.1.4 Constants**

27 The following constants are defined in the class `Object`.

28 **STDIN:** An implementation-defined readable instance of the class `IO`, which is used for  
29 reading conventional input.

30 **STDOUT:** An implementation-defined writable instance of the class `IO`, which is used for  
31 writing conventional output.

1       **STDERR:** An implementation-defined writable instance of the class `IO`, which is used for  
2       writing diagnostic output.

3       Other than these constants, for each built-in class or module, including the class `Object` itself,  
4       a conforming processor shall define a constant in the class `Object`, whose name is the name of  
5       the class or module, and whose value is the class or module itself.

#### 6   **15.2.1.5 Instance methods**

##### 7   **15.2.1.5.1 `Object#initialize`**

---

8       `initialize(*args)`

---

9       **Visibility:** private

10       **Behavior:** The method `initialize` is the default object initialization method, which is  
11       invoked when an instance is created (see 13.2.4). It returns an implementation-defined  
12       value.

13       If the class `Object` is not the root of the class inheritance tree, the method `initialize` shall be  
14       defined in the class which is the root of the class inheritance tree instead of in the class `Object`.

#### 15   **15.2.2 Module**

##### 16   **15.2.2.1 General description**

17       All modules are instances of the class `Module`. Therefore, behaviors defined in the class `Module`  
18       are shared by all modules.

19       The binary relation on the instances of the class `Module` denoted  $A \sqsubset B$  is defined as follows:

- 20       •  $B$  is a module, and  $B$  is included in  $A$  (see 13.1.3) or  
21       • Both  $A$  and  $B$  are instances of the class `Class`, and  $B$  is a superclass of  $A$ .

##### 22   **15.2.2.2 Direct superclass**

23       The class `Object`

##### 24   **15.2.2.3 Singleton methods**

##### 25   **15.2.2.3.1 `Module.constants`**

---

26       `Module.constants`

---

27       **Visibility:** public

28       **Behavior:**

- 1 a) Create an empty direct instance of the class `Array`. Let  $A$  be the instance.
- 2 b) Let  $C$  be the current class or module. Let  $L$  be the list which consists of the same  
3 elements as the list at the second element from the top of `[[class-module-list]]`, except  
4 the last element, which is the class `Object`.
- 5 Let  $CS$  be the set of classes which consists of  $C$  and all the superclasses of  $C$  except  
6 the class `Object`, but when  $C$  is the class `Object`, it shall be included in  $CS$ . Let  $MS$   
7 be the set of modules which consists of all the modules in the included module list of  
8 all classes in  $CS$ . Let  $CM$  be the union of  $L$ ,  $CS$  and  $MS$ .
- 9 c) For each class or module  $c$  in  $CM$ , and for each name  $N$  of a constant defined in  $c$ ,  
10 take the following steps:
- 11 1) Let  $S$  be either a new direct instance of the class `String` whose content is  $N$  or a  
12 direct instance of the class `Symbol` whose name is  $N$ . Which is chosen as the value  
13 of  $S$  is implementation-defined.
- 14 2) Unless  $A$  contains the element of the same name as  $S$ , when  $S$  is an instance of the  
15 class `Symbol`, or the same content as  $S$ , when  $S$  is an instance of the class `String`,  
16 insert  $S$  to  $A$ . The position where  $S$  is inserted is implementation-defined.
- 17 d) Return  $A$ .

#### 18 15.2.2.3.2 `Module.nesting`

---

19 `Module.nesting`

---

20 **Visibility:** public

21 **Behavior:** The method returns a new direct instance of the class `Array` which contains all  
22 but the last element of the list at the second element from the top of the `[[class-module-list]]`  
23 in the same order.

#### 24 15.2.2.4 Instance methods

##### 25 15.2.2.4.1 `Module#<`

---

26 `<(other)`

---

27 **Visibility:** public

28 **Behavior:** Let  $A$  be *other*. Let  $R$  be the receiver of the method.

- 29 a) If  $A$  is not an instance of the class `Module`, raise a direct instance of the class `TypeError`.
- 30 b) If  $A$  and  $R$  are the same object, return **false**.
- 31 c) If  $R \sqsubset A$ , return **true**.

1 d) If  $A \sqsubset R$ , return **false**.

2 e) Otherwise, return **nil**.

3 **15.2.2.4.2 Module#<=**

---

4 **<=( other )**

---

5 **Visibility:** public

6 **Behavior:**

7 a) If *other* and the receiver are the same object, return **true**.

8 b) Otherwise, the behavior is the same as the method  $<$  (see 15.2.2.4.1).

9 **15.2.2.4.3 Module#<=>**

---

10 **<=>( other )**

---

11 **Visibility:** public

12 **Behavior:** Let  $A$  be *other*. Let  $R$  be the receiver of the method.

13 a) If  $A$  is not an instance of the class `Module`, return **nil**.

14 b) If  $A$  and  $R$  are the same object, return an instance of the class `Integer` whose value  
15 is 0.

16 c) If  $R \sqsubset A$ , return an instance of the class `Integer` whose value is  $-1$ .

17 d) If  $A \sqsubset R$ , return an instance of the class `Integer` whose value is 1.

18 e) Otherwise, return **nil**.

19 **15.2.2.4.4 Module#==**

---

20 **==( other )**

---

21 **Visibility:** public

22 **Behavior:** Same as the method `==` of the module `Kernel` (see 15.3.1.3.1).

23 **15.2.2.4.5 Module#===**

---

1     ===(*object*)

---

2     **Visibility:** public

3     **Behavior:** Invoke the method `kind_of?` (see 15.3.1.3.26) of the module `Kernel` on *object*  
4     with the receiver as the only argument, and return the resulting value.

5     **15.2.2.4.6**   Module#>

---

6     >( *other* )

---

7     **Visibility:** public

8     **Behavior:** Let *A* be *other*. Let *R* be the receiver of the method.

9     a) If *A* is not an instance of the class `Module`, raise a direct instance of the class `TypeError`.

10    b) If *A* and *R* are the same object, return **false**.

11    c) If  $R \sqsubset A$ , return **false**.

12    d) If  $A \sqsubset R$ , return **true**.

13    e) Otherwise, return **nil**.

14    **15.2.2.4.7**   Module#>=

---

15    >=( *other* )

---

16    **Visibility:** public

17    **Behavior:**

18    a) If *other* and the receiver are the same object, return **true**.

19    b) Otherwise, the behavior is the same as the method `>` (see 15.2.2.4.6).

20    **15.2.2.4.8**   Module#alias\_method

---

21    alias\_method(*new\_name*, *aliased\_name*)

---

22    **Visibility:** private

23    **Behavior:** Let *C* be the receiver of the method.

- 1 a) Let  $N$  be the name designated by  $new\_name$ . Let  $A$  be the name designated by  
2  $aliased\_name$ .
- 3 b) Take steps d) through h) of 13.3.6, assuming that  $A$ ,  $C$ , and  $N$  in 13.3.6 to be  $A$ ,  $C$ ,  
4 and  $N$  in the above steps.
- 5 c) Return  $C$ .

#### 6 15.2.2.4.9 `Module#ancestors`

---

7 `ancestors`

---

8 **Visibility:** public

9 **Behavior:**

- 10 a) Create an empty direct instance  $A$  of the class `Array`.
- 11 b) Let  $C$  be the receiver of the method.
- 12 c) If  $C$  is a singleton class, the behavior is implementation-defined.
- 13 d) Otherwise, append  $C$  to the end of  $A$ .
- 14 e) Append each element of the included module list of  $C$  to  $A$  in the reverse order.
- 15 f) If  $C$  has a direct superclass, let new  $C$  be the direct superclass of the current  $C$ , and  
16 repeat from Step c).
- 17 g) Return  $A$ .

#### 18 15.2.2.4.10 `Module#append_features`

---

19 `append_features(module)`

---

20 **Visibility:** private

21 **Behavior:** Let  $L1$  and  $L2$  be the included module list of the receiver and  $module$  respec-  
22 tively.

- 23 a) If  $module$  and the receiver are the same object, the behavior is unspecified.
- 24 b) If the receiver is an element of  $L2$ , the behavior is implementation-defined.
- 25 c) Otherwise, for each module  $M$  in  $L1$ , in the same order in  $L1$ , take the following steps:
- 26 1) If  $M$  and  $module$  are the same object, the behavior is unspecified.
- 27 2) If  $M$  is not in  $L2$ , append  $M$  to the end of  $L2$ .

- 1 d) Append the receiver to *L2*.  
2 e) Return an implementation-defined value.

### 3 15.2.2.4.11 Module#attr

---

4 attr(*name*)

---

5 **Visibility:** private

6 **Behavior:** Invoke the method `attr_reader` of the class `Module` (see 15.2.2.4.13) on the  
7 receiver with *name* as the only argument, and return the resulting value.

### 8 15.2.2.4.12 Module#attr\_accessor

---

9 attr\_accessor(\**name\_list*)

---

10 **Visibility:** private

11 **Behavior:**

12 Let *C* be the method receiver.

13 a) For each element *E* of *name\_list*, take the following steps:

14 1) Let *N* be the name designated by *E*.

15 2) If *N* is not of the form *local-variable-identifier* or *constant-identifier*, raise a direct  
16 instance of the class `NameError` which has *E* as its name attribute.

17 3) Define an instance method in *C* as if by evaluating the following method definition  
18 at the location of the invocation. In the following method definition, *N* is *N*, and  
19 @*N* is the name which is *N* prefixed by “@”.

```
20         def N
21             @N
22         end
```

24 4) Define an instance method in *C* as if by evaluating the following method definition  
25 at the location of the invocation. In the following method definition, *N=* is the name  
26 *N* postfixed by =, and @*N* is the name which is *N* prefixed by “@”. The choice of the  
27 parameter name is arbitrary, and `val` is chosen only for the expository purpose.

```
28         def N=(val)
29             @N = val
30         end
```

32 b) Return an implementation-defined value.

1 **15.2.2.4.13 Module#attr\_reader**

---

2 attr\_reader(\*name\_list)

---

3 **Visibility:** private

4 **Behavior:** The method takes the same steps as the method `attr_accessor` [see 15.2.2.4.12,  
5 except Step a) 4)] of the class `Module`.

6 **15.2.2.4.14 Module#attr\_writer**

---

7 attr\_writer(\*name\_list)

---

8 **Visibility:** private

9 **Behavior:** The method takes the same steps as the method `attr_accessor` of the class  
10 `Module` (see 15.2.2.4.12), except Step a) 3).

11 **15.2.2.4.15 Module#class\_eval**

---

12 class\_eval(string=nil, &block)

---

13 **Visibility:** public

14 **Behavior:**

15 a) Let  $M$  be the receiver.

16 b) If  $block$  is given:

17 1) If  $string$  is given, raise a direct instance of the class `ArgumentError`.

18 2) Call  $block$  with implementation-defined arguments as described in 11.3.3, and let  $V$   
19 be the resulting value. A conforming processor shall modify the execution context  
20 just before 11.3.3 d) as follows:

21 • Create a new list which has the same members as those of the list at the top  
22 of `[[class-module-list]]`, and add  $M$  to the head of the newly created list. Push  
23 the list onto `[[class-module-list]]`.

24 • Push the receiver onto `[[self]]`.

25 • Push the public visibility onto `[[default-method-visibility]]`.

26 In 11.3.3 d) and e), a conforming processor shall ignore  $M$  which is added to the  
27 head of the top of `[[class-module-list]]` as described above, except when referring to  
28 the current class or module in a *method-definition* (see 13.3.1), an *alias-statement*  
29 (see 13.3.6), or an *undef-statement* (see 13.3.7).

- 1           3) Return  $V$ .
- 2       c) If  $block$  is not given:
- 3           1) If  $string$  is not an instance of the class `String`, the behavior is unspecified.
- 4           2) Let  $E$  be the execution context as it exists just before this method invoked.
- 5           3) Modify  $E$  as follows:
- 6               • Create a new list which has the same members as those of the list at the top
- 7               of `[[class-module-list]]`, and add  $M$  to the head of the newly created list. Push
- 8               the list onto `[[class-module-list]]`.
- 9               • Push the receiver onto `[[self]]`.
- 10              • Push the public visibility onto `[[default-method-visibility]]`.
- 11           4) Parse the content of  $string$  as a *program* (see 10.1). If it fails, raise a direct instance
- 12           of the class `SyntaxError`.
- 13           5) Evaluate the *program* within the execution context  $E$ . Let  $V$  be the resulting value
- 14           of the evaluation.
- 15           6) Restore the execution context  $E$  by removing the elements from the tops of `[[class-`
- 16           `module-list]]`, `[[self]]`, and `[[default-method-visibility]]`, even when an exception is
- 17           raised and not handled in c) 4) or c) 5).
- 18           7) Return  $V$ .

19       In Step c) 5), a local variable scope which corresponds to the *program* is considered as a

20       local variable scope which corresponds to a *block* in 9.2 d) 1).

#### 21 15.2.2.4.16 `Module#class_variable_defined?`

---

22       `class_variable_defined?(symbol)`

---

23       **Visibility:** public

24       **Behavior:** Let  $C$  be the receiver of the method.

- 25       a) Let  $N$  be the name designated by  $symbol$ .
- 26       b) If  $N$  is not of the form *class-variable-identifier*, raise a direct instance of the class
- 27       `NameError` which has  $symbol$  as its name attribute.
- 28       c) Search for a binding of the class variable with name  $N$  by taking steps b) through d)
- 29       of 11.5.4.5, assuming that  $C$  and  $N$  in 11.5.4.5 to be  $C$  and  $N$  in the above steps.
- 30       d) If a binding is found, return **true**.
- 31       e) Otherwise, return **false**.

1 **15.2.2.4.17** `Module#class_variable_get`

---

2 `class_variable_get( symbol )`

---

3 **Visibility:** implementation-defined

4 **Behavior:** Let  $C$  be the receiver of the method.

- 5 a) Let  $N$  be the name designated by *symbol*.
- 6 b) If  $N$  is not of the form *class-variable-identifier*, raise a direct instance of the class  
7 `NameError` which has *symbol* as its name attribute.
- 8 c) Search for a binding of the class variable with name  $N$  by taking steps b) through d)  
9 of 11.5.4.5, assuming that  $C$  and  $N$  in 11.5.4.5 to be  $C$  and  $N$  in the above steps.
- 10 d) If a binding is found, return the value of the binding.
- 11 e) Otherwise, raise a direct instance of the class `NameError` which has *symbol* as its name  
12 attribute.

13 **15.2.2.4.18** `Module#class_variable_set`

---

14 `class_variable_set( symbol, obj )`

---

15 **Visibility:** implementation-defined

16 **Behavior:** Let  $C$  be the receiver of the method.

- 17 a) Let  $N$  be the name designated by *symbol*.
- 18 b) If  $N$  is not of the form *class-variable-identifier*, raise a direct instance of the class  
19 `NameError` which has *symbol* as its name attribute.
- 20 c) Search for a binding of the class variable with name  $N$  by taking steps b) through d)  
21 of 11.5.4.5, assuming that  $C$  and  $N$  in 11.5.4.5 to be  $C$  and  $N$  in the above steps.
- 22 d) If a binding is found, replace the value of the binding with *obj*.
- 23 e) Otherwise, create a variable binding with name  $N$  and value *obj* in the set of bindings  
24 of class variables of  $C$ .
- 25 f) Return *obj*.

26 **15.2.2.4.19** `Module#class_variables`

---

1 `class_variables`

---

2 **Visibility:** public

3 **Behavior:** The method returns a new direct instance of the class `Array` which consists  
4 of names of all class variables of the receiver. These names are represented by direct  
5 instances of either the class `String` or the class `Symbol`. Which of those classes is chosen is  
6 implementation-defined.

7 **15.2.2.4.20 `Module#const_defined?`**

---

8 `const_defined?(symbol)`

---

9 **Visibility:** public

10 **Behavior:**

- 11 a) Let  $C$  be the receiver of the method.
- 12 b) Let  $N$  be the name designated by *symbol*.
- 13 c) If  $N$  is not of the form *constant-identifier*, raise a direct instance of the class `NameError`  
14 which has *symbol* as its name attribute.
- 15 d) If a binding with name  $N$  exists in the set of bindings of constants of  $C$ , return **true**.
- 16 e) Otherwise, return **false**.

17 **15.2.2.4.21 `Module#const_get`**

---

18 `const_get(symbol)`

---

19 **Visibility:** public

20 **Behavior:**

- 21 a) Let  $N$  be the name designated by *symbol*.
- 22 b) If  $N$  is not of the form *constant-identifier*, raise a direct instance of the class `NameError`  
23 which has *symbol* as its name attribute.
- 24 c) Search for a binding of a constant with name  $N$  from Step e) of 11.5.4.2, assuming that  
25  $C$  in 11.5.4.2 to be the receiver of the method.
- 26 d) If a binding is found, return the value of the binding.
- 27 e) Otherwise, return the value of the invocation of the method `const_missing` [see 11.5.4.2  
28 e) 1) i)].

#### 1 15.2.2.4.22 Module#const\_missing

---

2 `const_missing(symbol)`

---

3 **Visibility:** public

4 **Behavior:** The method `const_missing` is invoked when a binding of a constant does not  
5 exist on a constant reference (see 11.5.4.2).

6 When the method is invoked, take the following steps:

- 7 a) Take steps a) through c) of 15.2.2.4.20.  
8 b) Raise a direct instance of the class `NameError` which has *symbol* as its name attribute.

#### 9 15.2.2.4.23 Module#const\_set

---

10 `const_set(symbol, obj)`

---

11 **Visibility:** public

12 **Behavior:** Let *C* be the receiver of the method.

- 13 a) Let *N* be the name designated by *symbol*.  
14 b) If *N* is not of the form *constant-identifier*, raise a direct instance of the class `NameError`  
15 which has *symbol* as its name attribute.  
16 c) If a binding with name *N* exists in the set of bindings of constants of *C*, replace the  
17 value of the binding with *obj*.  
18 d) Otherwise, create a variable binding with *N* and value *obj* in the set of bindings of  
19 constants of *C*.  
20 e) Return *obj*.

#### 21 15.2.2.4.24 Module#constants

---

22 `constants`

---

23 **Visibility:** public

24 **Behavior:**

25 The method returns a new direct instance of the class `Array` which consists of names of all  
26 constants defined in the receiver. These names are represented by direct instances of either  
27 the class `String` or the class `Symbol`. Which of those classes is chosen is implementation-  
28 defined.

1 **15.2.2.4.25** `Module#extend_object`

---

2 `extend_object(object)`

---

3 **Visibility:** private

4 **Behavior:** Let  $S$  be the singleton class of *object*. Invoke the method `append_features` (see  
5 15.2.2.4.10) on the receiver with  $S$  as the only argument, and return the resulting value.

6 **15.2.2.4.26** `Module#extended`

---

7 `extended(object)`

---

8 **Visibility:** private

9 **Behavior:** The method returns `nil`.

10 NOTE The method `extended` is invoked in the method `extend` of the module `Kernel` (see 15.3.1.3.13).  
11 The method `extended` can be overridden to hook an invocation of the method `extend`.

12 **15.2.2.4.27** `Module#include`

---

13 `include(*module_list)`

---

14 **Visibility:** private

15 **Behavior:** Let  $C$  be the receiver of the method.

16 a) For each element  $A$  of *module\_list*, in the reverse order in *module\_list*, take the following  
17 steps:

18 1) If  $A$  is not an instance of the class `Module`, raise a direct instance of the class  
19 `TypeError`.

20 2) If  $A$  is an instance of the class `Class`, raise a direct instance of the class `TypeError`.

21 3) Invoke the method `append_features` (see 15.2.2.4.10) on  $A$  with  $C$  as the only  
22 argument.

23 4) Invoke the method `included` (see 15.2.2.4.29) on  $A$  with  $C$  as the only argument.

24 b) Return  $C$ .

25 **15.2.2.4.28** `Module#include?`

---

1 `include?( module )`

---

2 **Visibility:** public

3 **Behavior:** Let *C* be the receiver of the method.

4 a) If *module* is not an instance of the class `Module`, raise a direct instance of the class  
5 `TypeError`.

6 b) If *module* is an element of the included module list of *C*, return **true**.

7 c) Otherwise, if *C* is an instance of the class `Class`, and if *module* is an element of the  
8 included module list of one of the superclasses of *C*, then return **true**.

9 d) Otherwise, return **false**.

#### 10 **15.2.2.4.29 Module#included**

---

11 `included( module )`

---

12 **Visibility:** private

13 **Behavior:** The method returns **nil**.

14 **NOTE** The method `included` is invoked in the method `include` of the class `Module` (see 15.2.2.4.27).  
15 The method `included` can be overridden to hook an invocation of the method `include`.

#### 16 **15.2.2.4.30 Module#included\_modules**

---

17 `included_modules`

---

18 **Visibility:** public

19 **Behavior:** Let *C* be the receiver of the method.

20 a) Create an empty direct instance *A* of the class `Array`.

21 b) Append each element of the included module list of *C*, in the reverse order, to *A*.

22 c) If *C* is an instance of the class `Class`, and if *C* has a direct superclass, then let new *C*  
23 be the direct superclass of the current *C*, and repeat from Step b).

24 d) Otherwise, return *A*.

#### 25 **15.2.2.4.31 Module#initialize**

---

1     `initialize(&block)`

---

2     **Visibility:** private

3     **Behavior:**

4     a) If *block* is given, take step b) of the method `class_eval` of the class `Module` (see  
5       15.2.2.4.15), assuming that *block* in 15.2.2.4.15 to be *block* given to this method.

6     b) Return an implementation-defined value.

7     **15.2.2.4.32 Module#initialize\_copy**

---

8     `initialize_copy(original)`

---

9     **Visibility:** private

10    **Behavior:**

11    a) Invoke the instance method `initialize_copy` defined in the module `Kernel` on the  
12       receiver with *original* as the argument.

13    b) If the receiver is associated with a singleton class, let  $E_o$  be the singleton class, and  
14       take the following steps:

15       1) Create a singleton class whose direct superclass is the direct superclass of  $E_o$ . Let  
16           $E_n$  be the singleton class.

17       2) For each binding  $B_{v1}$  of the constants of  $E_o$ , create a variable binding with the  
18          same name and value as  $B_{v1}$  in the set of bindings of constants of  $E_n$ .

19       3) For each binding  $B_{v2}$  of the class variables of  $E_o$ , create a variable binding with  
20          the same name and value as  $B_{v2}$  in the set of bindings of class variables of  $E_n$ .

21       4) For each binding  $B_m$  of the instance methods of  $E_o$ , create a method binding with  
22          the same name and value as  $B_m$  in the set of bindings of instance methods of  $E_n$ .

23       5) Associate the receiver with  $E_n$ .

24    c) If the receiver is an instance of the class `Class`:

25       1) If *original* has a direct superclass, set the direct superclass of the receiver to the  
26          direct superclass of *original*.

27       2) Otherwise, the behavior is unspecified.

28    d) Append each element of the included module list of *original*, in the same order, to the  
29       included module list of the receiver.

- 1 e) For each binding  $B_{v3}$  of the constants of *original*, create a variable binding with the  
2 same name and value as  $B_{v3}$  in the set of bindings of constants of the receiver.
- 3 f) For each binding  $B_{v4}$  of the class variables of *original*, create a variable binding with  
4 the same name and value as  $B_{v4}$  in the set of bindings of class variables of the receiver.
- 5 g) For each binding  $B_{m2}$  of the instance methods of *original*, create a method binding  
6 with the same name and value as  $B_{m2}$  in the set of bindings of instance methods of the  
7 receiver.
- 8 h) Return an implementation-defined value.

#### 9 15.2.2.4.33 Module#instance\_methods

---

```
10 instance_methods( include_super=true )
```

---

11 **Visibility:** public

12 **Behavior:** Let  $C$  be the receiver of the method.

- 13 a) Create an empty direct instance  $A$  of the class **Array**.
- 14 b) Let  $I$  be the set of bindings of instance methods of  $C$ . For each binding  $B$  of  $I$ , let  $N$   
15 be the name of  $B$ , and let  $V$  be the value of  $B$ , and take the following steps:
  - 16 1) If  $V$  is undef, or the visibility of  $V$  is private, skip the next two steps.
  - 17 2) Let  $S$  be either a new direct instance of the class **String** whose content is  $N$  or a  
18 direct instance of the class **Symbol** whose name is  $N$ . Which is chosen as the value  
19 of  $S$  is implementation-defined.
  - 20 3) Unless  $A$  contains the element of the same name (if  $S$  is an instance of the class  
21 **Symbol**) or the same content (if  $S$  is an instance of the class **String**) as  $S$ , append  
22  $S$  to  $A$ .
- 23 c) If *include\_super* is a trueish object:
  - 24 1) For each module  $M$  in included module list of  $C$ , take step b), assuming that  $C$   
25 in that step to be  $M$ .
  - 26 2) If  $C$  does not have a direct superclass, return  $A$ .
  - 27 3) Let new  $C$  be the direct superclass of  $C$ .
  - 28 4) Repeat from Step b).
- 29 d) Return  $A$ .

#### 30 15.2.2.4.34 Module#method\_defined?

---

1 `method_defined?( symbol )`

---

2 **Visibility:** public

3 **Behavior:** Let *C* be the receiver of the method.

4 a) Let *N* be the name designated by *symbol*.

5 b) Search for a binding of an instance method named *N* starting from *C* as described in  
6 13.3.4.

7 c) If a binding is found and its value is not undef, return **true**.

8 d) Otherwise, return **false**.

9 **15.2.2.4.35 Module#module\_eval**

---

10 `module_eval( string=nil, &block )`

---

11 **Visibility:** public

12 **Behavior:** Same as the method `class_eval` (see 15.2.2.4.15).

13 **15.2.2.4.36 Module#private**

---

14 `private( *symbol_list )`

---

15 **Visibility:** private

16 **Behavior:** Same as the method `public` (see 15.2.2.4.38), except to let *NV* be the private  
17 visibility in 15.2.2.4.38 a).

18 **15.2.2.4.37 Module#protected**

---

19 `protected( *symbol_list )`

---

20 **Visibility:** private

21 **Behavior:** Same as the method `public` (see 15.2.2.4.38), except to let *NV* be the protected  
22 visibility in 15.2.2.4.38 a).

23 **15.2.2.4.38 Module#public**

---

1     `public(*symbol_list)`

---

2     **Visibility:** private

3     **Behavior:** Let *C* be the receiver of the method.

4     a) Let *NV* be the public visibility.

5     b) If the length of *symbol\_list* is 0, change the current visibility to *NV* and return *C*.

6     c) Otherwise, for each element *S* of *symbol\_list*, take the following steps:

7         1) Let *N* be the name designated by *S*.

8         2) Search for a method binding with name *N* starting from *C* as described in 13.3.4.

9         3) If a binding is found and its value is not undef, let *V* the value of the binding.

10        4) Otherwise, raise a direct instance of the class `NameError` which has *S* as its name  
11        attribute.

12        5) If *C* is the class or module in which the binding is found, change the visibility of  
13        *V* to *NV*.

14        6) Otherwise, define an instance method in *C* as if by evaluating the following method  
15        definition. In the definition, *N* is *N*. The choice of the parameter name is arbitrary,  
16        and `args` is chosen only for the expository purpose.

```
17                    def N(*args)
18                        super
19                        end
```

21     The attributes of the method created by the above definition are initialized as  
22     follows:

23     i) The class module list is the element at the top of `[[class-module-list]]`.

24     ii) The defined name is the defined name of *V*.

25     iii) The visibility is *NV*.

26     d) Return *C*.

#### 27 **15.2.2.4.39** `Module#remove_class_variable`

---

28     `remove_class_variable(symbol)`

---

1 **Visibility:** implementation-defined

2 **Behavior:** Let  $C$  be the receiver of the method.

3 a) Let  $N$  be the name designated by  $symbol$ .

4 b) If  $N$  is not of the form *class-variable-identifier*, raise a direct instance of the class  
5 `NameError` which has  $symbol$  as its name attribute.

6 c) If a binding with name  $N$  exists in the set of bindings of class variables of  $C$ , let  $V$  be  
7 the value of the binding.

8 1) Remove the binding from the set of bindings of class variables of  $C$ .

9 2) Return  $V$ .

10 d) Otherwise, raise a direct instance of the class `NameError` which has  $symbol$  as its name  
11 attribute.

#### 12 15.2.2.4.40 `Module#remove_const`

---

13 `remove_const( symbol )`

---

14 **Visibility:** private

15 **Behavior:** Let  $C$  be the receiver of the method.

16 a) Let  $N$  be the name designated by  $symbol$ .

17 b) If  $N$  is not of the form *constant-identifier*, raise a direct instance of the class `NameError`  
18 which has  $symbol$  as its name attribute.

19 c) If a binding with name  $N$  exists in the set of bindings of constants of  $C$ , let  $V$  be the  
20 value of the binding.

21 1) Remove the binding from the set of bindings of constants of  $C$ .

22 2) Return  $V$ .

23 d) Otherwise, raise a direct instance of the class `NameError` which has  $symbol$  as its name  
24 attribute.

#### 25 15.2.2.4.41 `Module#remove_method`

---

26 `remove_method(*symbol_list)`

---

27 **Visibility:** private

28 **Behavior:** Let  $C$  be the receiver of the method.

- 1 a) For each element *S* of *symbol\_list*, in the order in the list, take the following steps:
  - 2 1) Let *N* be the name designated by *S*.
  - 3 2) If a binding with name *N* exists in the set of bindings of instance methods of *C*,  
4 and if the value of the binding is not undef, then remove the binding from the set.
  - 5 3) Otherwise, raise a direct instance of the class `NameError` which has *S* as its name  
6 attribute. In this case, the remaining elements of *symbol\_list* are not processed.
- 7 b) Return *C*.

#### 8 **15.2.2.4.42 Module#undef\_method**

---

9 `undef_method(*symbol_list)`

---

10 **Visibility:** private

11 **Behavior:** Let *C* be the receiver of the method.

- 12 a) For each element *S* of *symbol\_list*, in the order in the list, take the following steps:
  - 13 1) Let *N* be the name designated by *S*.
  - 14 2) Take steps a) 3) and a) 4) of 13.3.7, assuming that *C* and *N* in 13.3.7 to be *C* and  
15 *N* in the above steps, respectively.
- 16 b) Return *C*.

### 17 **15.2.3 Class**

#### 18 **15.2.3.1 General description**

19 All classes are instances of the class `Class`. Therefore, behaviors defined in the class `Class` are  
20 shared by all classes.

21 The instance methods `append_features` and `extend_object` of the class `Class` shall be unde-  
22 fined by invoking the method `undef_method` (see 15.2.2.4.42) on the class `Class` with instances  
23 of the class `Symbol` whoses names are “append\_features” and “extend\_object” as the arguments.

24 **NOTE** The instance methods `append_features` and `extend_object` are methods for modules. These  
25 methods are therefore undefined in the class `Class`, whose instances do not represent modules, but classes.

#### 26 **15.2.3.2 Direct superclass**

27 The class `Module`

#### 28 **15.2.3.3 Instance methods**

##### 29 **15.2.3.3.1 Class#initialize**

---

1 `initialize( superclass=Object, &block )`

---

2 **Visibility:** private

3 **Behavior:**

- 4 a) If the receiver has its direct superclass, or is the root of the class inheritance tree, then  
5 raise a direct instance of the class `TypeError`.
- 6 b) If *superclass* is not an instance of the class `Class`, raise a direct instance of the class  
7 `TypeError`.
- 8 c) If *superclass* is a singleton class or the class `Class`, the behavior is unspecified.
- 9 d) Set the direct superclass of the receiver to *superclass*.
- 10 e) Create a singleton class, and associate it with the receiver. The singleton class shall  
11 have the singleton class of *superclass* as one of its superclasses.
- 12 f) If *block* is given, take step b) of the method `class_eval` of the class `Module` (see  
13 15.2.2.4.15), assuming that *block* in 15.2.2.4.15 to be *block* given to this method.
- 14 g) Return an implementation-defined value.

15 **15.2.3.3.2 Class#initialize\_copy**

---

16 `initialize_copy( original )`

---

17 **Visibility:** private

18 **Behavior:**

- 19 a) If the direct superclass of the receiver has already been set, or if the receiver is the root  
20 of the class inheritance tree, then raise a direct instance of the class `TypeError`.
- 21 b) If the receiver is a singleton class, raise a direct instance of the class `TypeError`.
- 22 c) Invoke the instance method `initialize_copy` defined in the class `Module` on the re-  
23 ceiver with *original* as the argument.
- 24 d) Return an implementation-defined value.

25 **15.2.3.3.3 Class#new**

---

26 `new( *args, &block )`

---

27 **Visibility:** public

1     **Behavior:**

- 2     a) If the receiver is a singleton class, raise a direct instance of the class `TypeError`.
- 3     b) Create a direct instance of the receiver which has no bindings of instance variables.  
4         Let *O* be the newly created instance.
- 5     c) Invoke the method `initialize` on *O* with all the elements of *args* as arguments and  
6         *block* as the block.
- 7     d) Return *O*.

8     **15.2.3.3.4 Class#superclass**

---

9     **superclass**

---

10    **Visibility:** public

11    **Behavior:** Let *C* be the receiver of the method.

- 12    a) If *C* is a singleton class, return an implementation-defined value.
- 13    b) If *C* does not have a direct superclass, return **nil**.
- 14    c) Otherwise, return the direct superclass of *C*.

15    **15.2.4 NilClass**

16    **15.2.4.1 General description**

17    The class `NilClass` has only one instance **nil** (see 6.6).

18    Instances of the class `NilClass` shall not be created by the method `new` of the class `NilClass`.  
19    Therefore, the singleton method `new` of the class `NilClass` shall be undefined, by invoking the  
20    method `undef_method` (see 15.2.2.4.42) on the singleton class of the class `NilClass` with a direct  
21    instance of the class `Symbol` whose name is “new” as the argument.

22    **15.2.4.2 Direct superclass**

23    The class `Object`

24    **15.2.4.3 Instance methods**

25    **15.2.4.3.1 NilClass#&**

---

26    **&(other)**

---

27    **Visibility:** public

28    **Behavior:** The method returns **false**.

1 **15.2.4.3.2 NilClass#^**

---

2 `^(other)`

---

3 **Visibility:** public

4 **Behavior:**

5 a) If *other* is a falseish object, return **false**.

6 b) Otherwise, return **true**.

7 **15.2.4.3.3 NilClass#|**

---

8 `|(other)`

---

9 **Visibility:** public

10 **Behavior:**

11 a) If *other* is a falseish object, return **false**.

12 b) Otherwise, return **true**.

13 **15.2.4.3.4 NilClass#nil?**

---

14 `nil?`

---

15 **Visibility:** public

16 **Behavior:** The method returns **true**.

17 **15.2.4.3.5 NilClass#to\_s**

---

18 `to_s`

---

19 **Visibility:** public

20 **Behavior:** The method creates an empty direct instance of the class `String`, and returns  
21 this instance.

22 **15.2.5 TrueClass**

23 **15.2.5.1 General description**

24 The class `TrueClass` has only one instance **true** (see 6.6).

1 Instances of the class `TrueClass` shall not be created by the method `new` of the class `TrueClass`.  
2 Therefore, the singleton method `new` of the class `TrueClass` shall be undefined, by invoking the  
3 method `undef_method` (see 15.2.2.4.42) on the singleton class of the class `TrueClass` with a  
4 direct instance of the class `Symbol` whose name is “new” as the argument.

### 5 **15.2.5.2 Direct superclass**

6 The class `Object`

### 7 **15.2.5.3 Instance methods**

#### 8 **15.2.5.3.1 `TrueClass#&`**

---

9 `&(other)`

---

10 **Visibility:** public

11 **Behavior:**

12 a) If *other* is a falseish object, return **false**.

13 b) Otherwise, return **true**.

#### 14 **15.2.5.3.2 `TrueClass#^`**

---

15 `^(other)`

---

16 **Visibility:** public

17 **Behavior:**

18 a) If *other* is a falseish object, return **true**.

19 b) Otherwise, return **false**.

#### 20 **15.2.5.3.3 `TrueClass#to_s`**

---

21 `to_s`

---

22 **Visibility:** public

23 **Behavior:** The method creates a direct instance of the class `String`, the content of which  
24 is “true”, and returns this instance.

#### 25 **15.2.5.3.4 `TrueClass#|`**

---

1 | (*other*)

---

2 | **Visibility:** public

3 | **Behavior:** The method returns **true**.

## 4 | 15.2.6 FalseClass

### 5 | 15.2.6.1 General description

6 | The class `FalseClass` has only one instance **false** (see 6.6).

7 | Instances of the class `FalseClass` shall not be created by the method `new` of the class `FalseClass`.  
8 | Therefore, the singleton method `new` of the class `FalseClass` shall be undefined, by invoking  
9 | the method `undef_method` (see 15.2.2.4.42) on the singleton class of the class `FalseClass` with  
10 | a direct instance of the class `Symbol` whose name is “new” as the argument.

### 11 | 15.2.6.2 Direct superclass

12 | The class `Object`

### 13 | 15.2.6.3 Instance methods

#### 14 | 15.2.6.3.1 `FalseClass#&`

---

15 | *&(other)*

---

16 | **Visibility:** public

17 | **Behavior:** The method returns **false**.

#### 18 | 15.2.6.3.2 `FalseClass#^`

---

19 | *^(other)*

---

20 | **Visibility:** public

21 | **Behavior:**

22 | a) If *other* is a falseish object, return **false**.

23 | b) Otherwise, return **true**.

#### 24 | 15.2.6.3.3 `FalseClass#to_s`

---

1        `to_s`

---

2        **Visibility:** public

3        **Behavior:** The method creates a direct instance of the class `String`, the content of which  
4        is “false”, and returns this instance.

5        **15.2.6.3.4 FalseClass#|**

---

6        `| (other)`

---

7        **Visibility:** public

8        **Behavior:**

9        a) If *other* is a falseish object, return **false**.

10       b) Otherwise, return **true**.

11       **15.2.7 Numeric**

12       **15.2.7.1 General description**

13       Instances of the class `Numeric` represent numbers. The class `Numeric` is the superclass of all the  
14       other built-in classes which represent numbers.

15       The notation “the value of the instance *N* of the class `Numeric`” means the number represented  
16       by *N*.

17       **15.2.7.2 Direct superclass**

18       The class `Object`

19       **15.2.7.3 Included modules**

20       The following module is included in the class `Numeric`.

- 21       • `Comparable`

22       **15.2.7.4 Instance methods**

23       **15.2.7.4.1 Numeric#+@**

---

24       `+@`

---

25       **Visibility:** public

26       **Behavior:** The method returns the receiver.

1 **15.2.7.4.2 Numeric#-@**

---

2 -@

---

3 **Visibility:** public

4 **Behavior:**

5 a) Invoke the method `coerce` on the receiver with an instance of the class `Integer` whose  
6 value is 0 as the only argument. Let  $V$  be the resulting value.

7 1) If  $V$  is an instance of the class `Array` which contains two elements, let  $F$  and  $S$   
8 be the first and the second element of  $V$  respectively.

9 i) Invoke the method `-` on  $F$  with  $S$  as the only argument.

10 ii) Return the resulting value.

11 2) Otherwise, raise a direct instance of the class `TypeError`.

12 **15.2.7.4.3 Numeric#abs**

---

13 `abs`

---

14 **Visibility:** public

15 **Behavior:**

16 a) Invoke the method `<` on the receiver with an instance of the class `Integer` whose value  
17 is 0 as an argument.

18 b) If this invocation results in a trueish object, invoke the method `-@` on the receiver and  
19 return the resulting value.

20 c) Otherwise, return the receiver.

21 **15.2.7.4.4 Numeric#coerce**

---

22 `coerce(other)`

---

23 **Visibility:** public

24 **Behavior:**

25 a) If the class of the receiver and the class of `other` are the same class, let  $X$  and  $Y$  be  
26 `other` and the receiver, respectively.

- 1       b) Otherwise, let *X* and *Y* be instances of the class `Float` which are converted from *other*  
2       and the receiver, respectively. *other* and the receiver are converted as follows:
- 3           1) Let *O* be *other* or the receiver.
- 4           2) If *O* is an instance of the class `Float`, let *F* be *O*.
- 5           3) Otherwise:
- 6               i) If an invocation of the method `respond_to?` on *O* with a direct instance of  
7               the class `Symbol` whose name is `to_f` as the argument results in a falseish  
8               object, raise a direct instance of the class `TypeError`.
- 9               ii) Invoke the method `to_f` on *O* with no arguments, and let *F* be the resulting  
10              value.
- 11              iii) If *F* is not an instance of the class `Float`, raise a direct instance of the class  
12              `TypeError`.
- 13           4) If the value of *F* is NaN, the behavior is unspecified.
- 14           5) The converted value of *O* is *F*.
- 15       c) Create a direct instance of the class `Array` which consists of two elements: the first is  
16       *X*; the second is *Y*.
- 17       d) Return the instance of the class `Array`.

## 18 **15.2.8 Integer**

### 19 **15.2.8.1 General description**

20 Instances of the class `Integer` represent integers. The ranges of these integers are unbounded.  
21 However the actual values computable depend on resource limitations, and the behavior when  
22 the resource limits are exceeded is implementation-defined.

23 Instances of the class `Integer` shall not be created by the method `new` of the class `Integer`.  
24 Therefore, the singleton method `new` of the class `Integer` shall be undefined, by invoking the  
25 method `undef_method` (see 15.2.2.4.42) on the singleton class of the class `Integer` with a direct  
26 instance of the class `Symbol` whose name is “new” as the argument.

27 Subclasses of the class `Integer` may be defined as built-in classes. In this case:

- 28 • The class `Integer` shall not have its direct instances. Instead of a direct instance of the  
29 class `Integer`, a direct instance of a subclass of the class `Integer` shall be created.
- 30 • Instance methods of the class `Integer` need not be defined in the class `Integer` itself if the  
31 instance methods are defined in all subclasses of the class `Integer`.
- 32 • For each subclass of the class `Integer`, the ranges of the values of its instances may be  
33 bounded.

## 1 15.2.8.2 Direct superclass

2 The class `Numeric`

## 3 15.2.8.3 Instance methods

### 4 15.2.8.3.1 `Integer#+`

---

5 `+(other)`

---

6 **Visibility:** public

7 **Behavior:**

8 a) If *other* is an instance of the class `Integer`, return an instance of the class `Integer`  
9 whose value is the sum of the values of the receiver and *other*.

10 b) If *other* is an instance of the class `Float`, let *R* be the value of the receiver as a  
11 floating-point number.

12 Return a direct instance of the class `Float` whose value is the sum of *R* and the value  
13 of *other*.

14 c) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument.  
15 Let *V* be the resulting value.

16 1) If *V* is an instance of the class `Array` which contains two elements, let *F* and *S*  
17 be the first and the second element of *V* respectively.

18 i) Invoke the method `+` on *F* with *S* as the only argument.

19 ii) Return the resulting value.

20 2) Otherwise, raise a direct instance of the class `TypeError`.

### 21 15.2.8.3.2 `Integer#-`

---

22 `-(other)`

---

23 **Visibility:** public

24 **Behavior:**

25 a) If *other* is an instance of the class `Integer`, return an instance of the class `Integer`  
26 whose value is the result of subtracting the value of *other* from the value of the receiver.

27 b) If *other* is an instance of the class `Float`, let *R* be the value of the receiver as a  
28 floating-point number.

1           Return a direct instance of the class `Float` whose value is the result of subtracting the  
2           value of *other* from *R*.

3           c) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument.  
4           Let *V* be the resulting value.

5           1) If *V* is an instance of the class `Array` which contains two elements, let *F* and *S*  
6           be the first and the second element of *V* respectively.

7           i) Invoke the method `-` on *F* with *S* as the only argument.

8           ii) Return the resulting value.

9           2) Otherwise, raise a direct instance of the class `TypeError`.

### 10 **15.2.8.3.3 Integer#\***

---

11           \*(*other*)

---

12           **Visibility:** public

13           **Behavior:**

14           a) If *other* is an instance of the class `Integer`, return an instance of the class `Integer`  
15           whose value is the result of multiplication of the values of the receiver and *other*.

16           b) If *other* is an instance of the class `Float`, let *R* be the value of the receiver as a  
17           floating-point number.

18           Return a direct instance of the class `Float` whose value is the result of multiplication  
19           of *R* and the value of *other*.

20           c) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument.  
21           Let *V* be the resulting value.

22           1) If *V* is an instance of the class `Array` which contains two elements, let *F* and *S*  
23           be the first and the second element of *V* respectively.

24           i) Invoke the method `*` on *F* with *S* as the only argument.

25           ii) Return the resulting value.

26           2) Otherwise, raise a direct instance of the class `TypeError`.

### 27 **15.2.8.3.4 Integer#/**

---

28           /(*other*)

---

1     **Visibility:** public

2     **Behavior:**

3     a) If *other* is an instance of the class **Integer**:

4         1) If the value of *other* is 0, raise a direct instance of the class **ZeroDivisionError**.

5         2) Otherwise, let *n* be the value of the receiver divided by the value of *other*. Return  
6             an instance of the class **Integer** whose value is the largest integer smaller than or  
7             equal to *n*.

8             NOTE The behavior is the same even if the receiver has a negative value. For example,  
9             -5 / 2 returns -3.

10     b) Otherwise, invoke the method **coerce** on *other* with the receiver as the only argument.  
11         Let *V* be the resulting value.

12         1) If *V* is an instance of the class **Array** which contains two elements, let *F* and *S*  
13             be the first and the second element of *V* respectively.

14             i) Invoke the method / on *F* with *S* as the only argument.

15             ii) Return the resulting value.

16         2) Otherwise, raise a direct instance of the class **TypeError**.

17 **15.2.8.3.5 Integer#%**

---

18     %( *other* )

---

19     **Visibility:** public

20     **Behavior:**

21     a) If *other* is an instance of the class **Integer**:

22         1) If the value of *other* is 0, raise a direct instance of the class **ZeroDivisionError**.

23         2) Otherwise, let *x* and *y* be the values of the receiver and *other*.

24             i) Let *t* be the largest integer smaller than or equal to *x* divided by *y*.

25             ii) Let *m* be  $x - t \times y$ .

26             iii) Otherwise, return an instance of the class **Integer** whose value is *m*.

27     b) Otherwise, invoke the method **coerce** on *other* with the receiver as the only argument.  
28         Let *V* be the resulting value.

- 1           1) If  $V$  is an instance of the class `Array` which contains two elements, let  $F$  and  $S$   
2           be the first and the second element of  $V$  respectively.
- 3           i) Invoke the method `%` on  $F$  with  $S$  as the only argument.
- 4           ii) Return the resulting value.
- 5           2) Otherwise, raise a direct instance of the class `TypeError`.

6 **15.2.8.3.6 Integer#<=>**

---

7           <=>( *other* )

---

8           **Visibility:** public

9           **Behavior:**

- 10          a) If *other* is an instance of the class `Integer`:
- 11           1) If the value of the receiver is larger than the value of *other*, return an instance of  
12           the class `Integer` whose value is 1.
- 13           2) If the values of the receiver and *other* are the same integer, return an instance of  
14           the class `Integer` whose value is 0.
- 15           3) If the value of the receiver is smaller than the value of *other*, return an instance  
16           of the class `Integer` whose value is  $-1$ .
- 17          b) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument.  
18           Let  $V$  be the resulting value.
- 19           1) If  $V$  is an instance of the class `Array` which contains two elements, let  $F$  and  $S$   
20           be the first and the second element of  $V$  respectively.
- 21           i) Invoke the method `<=>` on  $F$  with  $S$  as the only argument.
- 22           ii) If this invocation does not result in an instance of the class `Integer`, the  
23           behavior is unspecified.
- 24           iii) Otherwise, return the value of this invocation.
- 25           2) Otherwise, return `nil`.

26 **15.2.8.3.7 Integer#==**

---

27           ==( *other* )

---

28           **Visibility:** public

1     **Behavior:**

2     a) If *other* is an instance of the class `Integer`:

3         1) If the values of the receiver and *other* are the same integer, return **true**.

4         2) Otherwise, return **false**.

5     b) Otherwise, invoke the method `==` on *other* with the receiver as the argument. Return  
6         the resulting value of this invocation.

7     **15.2.8.3.8 Integer#~**

---

8     ~

---

9     **Visibility:** public

10    **Behavior:** The method returns an instance of the class `Integer` whose two's complement  
11    representation is the one's complement of the two's complement representation of the re-  
12    ceiver.

13    **15.2.8.3.9 Integer#&**

---

14    &( *other* )

---

15    **Visibility:** public

16    **Behavior:**

17    a) If *other* is not an instance of the class `Integer`, the behavior is unspecified.

18    b) Otherwise, return an instance of the class `Integer` whose two's complement represen-  
19    tation is the bitwise AND of the two's complement representations of the receiver and  
20    *other*.

21    **15.2.8.3.10 Integer#|**

---

22    |( *other* )

---

23    **Visibility:** public

24    **Behavior:**

25    a) If *other* is not an instance of the class `Integer`, the behavior is unspecified.

26    b) Otherwise, return an instance of the class `Integer` whose two's complement repre-  
27    sentation is the bitwise inclusive OR of the two's complement representations of the  
28    receiver and *other*.

1 **15.2.8.3.11 Integer#^**

---

2 `^(other)`

---

3 **Visibility:** public

4 **Behavior:**

- 5 a) If *other* is not an instance of the class `Integer`, the behavior is unspecified.
- 6 b) Otherwise, return an instance of the class `Integer` whose two's complement repre-  
7 sentation is the bitwise exclusive OR of the two's complement representations of the  
8 receiver and *other*.

9 **15.2.8.3.12 Integer#<<**

---

10 `<<(other)`

---

11 **Visibility:** public

12 **Behavior:**

- 13 a) If *other* is not an instance of the class `Integer`, the behavior is unspecified.
- 14 b) Otherwise, let *x* and *y* be the values of the receiver and *other*.
- 15 c) Return an instance of the class `Integer` whose value is the largest integer smaller than  
16 or equal to  $x \times 2^y$ .

17 **15.2.8.3.13 Integer#>>**

---

18 `>>(other)`

---

19 **Visibility:** public

20 **Behavior:**

- 21 a) If *other* is not an instance of the class `Integer`, the behavior is unspecified.
- 22 b) Otherwise, let *x* and *y* be the values of the receiver and *other*.
- 23 c) Return an instance of the class `Integer` whose value is the largest integer smaller than  
24 or equal to  $x \times 2^{-y}$ .

25 **15.2.8.3.14 Integer#ceil**

---

1     `ceil`

---

2     **Visibility:** public

3     **Behavior:** The method returns the receiver.

4     **15.2.8.3.15 Integer#downto**

---

5     `downto( num, &block )`

---

6     **Visibility:** public

7     **Behavior:**

8     a) If *num* is not an instance of the class `Integer`, or *block* is not given, the behavior is  
9       unspecified.

10    b) Let *i* be the value of the receiver.

11    c) If *i* is smaller than the value of *num*, return the receiver.

12    d) Call *block* with an instance of the class `Integer` whose value is *i*.

13    e) Decrement *i* by 1 and continue processing from Step c).

14    **15.2.8.3.16 Integer#eql?**

---

15    `eql?( other )`

---

16    **Visibility:** public

17    **Behavior:**

18    a) If *other* is not an instance of the class `Integer`, return **false**.

19    b) Otherwise, invoke the method `==` on *other* with the receiver as the argument.

20    c) If this invocation results in a trueish object, return **true**. Otherwise, return **false**.

21    **15.2.8.3.17 Integer#floor**

---

22    `floor`

---

23    **Visibility:** public

24    **Behavior:** The method returns the receiver.

1 **15.2.8.3.18 Integer#hash**

---

2 **hash**

---

3 **Visibility:** public

4 **Behavior:** The method returns an implementation-defined instance of the class `Integer`,  
5 which satisfies the following condition:

6 a) Let  $I_1$  and  $I_2$  be instances of the class `Integer`.

7 b) Let  $H_1$  and  $H_2$  be the resulting values of invocations of the method `hash` on  $I_1$  and  $I_2$ ,  
8 respectively.

9 c) The values of  $H_1$  and  $H_2$  shall be the same integer, if the values of  $I_1$  and  $I_2$  are the  
10 same integer.

11 **15.2.8.3.19 Integer#next**

---

12 **next**

---

13 **Visibility:** public

14 **Behavior:** The method returns an instance of the class `Integer`, whose value is the value  
15 of the receiver plus 1.

16 **15.2.8.3.20 Integer#round**

---

17 **round**

---

18 **Visibility:** public

19 **Behavior:** The method returns the receiver.

20 **15.2.8.3.21 Integer#succ**

---

21 **succ**

---

22 **Visibility:** public

23 **Behavior:** Same as the method `next` (see 15.2.8.3.19).

24 **15.2.8.3.22 Integer#times**

---

1 `times(&block)`

---

2 **Visibility:** public

3 **Behavior:**

- 4 a) If *block* is not given, the behavior is unspecified.
- 5 b) Let *i* be 0.
- 6 c) If *i* is larger than or equal to the value of the receiver, return the receiver.
- 7 d) Call *block* with an instance of the class `Integer` whose value is *i* as an argument.
- 8 e) Increment *i* by 1 and continue processing from Step c).

9 **15.2.8.3.23 Integer#to\_f**

---

10 `to_f`

---

11 **Visibility:** public

12 **Behavior:** The method returns a direct instance of the class `Float` whose value is the  
13 value of the receiver as a floating-point number.

14 **15.2.8.3.24 Integer#to\_i**

---

15 `to_i`

---

16 **Visibility:** public

17 **Behavior:** The method returns the receiver.

18 **15.2.8.3.25 Integer#to\_s**

---

19 `to_s`

---

20 **Visibility:** public

21 **Behavior:** The method returns a direct instance of the class `String` whose content satisfy  
22 the following conditions:

- 23 • If the value of the receiver is negative, the first character is the character “-” (0x2d).
- 24 • The sequence *R* of the rest of characters represents the magnitude *M* of the value of  
25 the receiver in base 10. If *M* is 0, *R* is a single “0”. Otherwise, the first character of  
26 *R* is not “0”.

1       EXAMPLE 1   123.to\_s returns "123".

2       EXAMPLE 2   -123.to\_s returns "-123".

### 3   15.2.8.3.26   Integer#truncate

---

4       truncate

---

5       **Visibility:** public

6       **Behavior:** The method returns the receiver.

### 7   15.2.8.3.27   Integer#upto

---

8       upto(*num*, &*block*)

---

9       **Visibility:** public

10      **Behavior:**

- 11      a) If *num* is not an instance of the class `Integer`, or *block* is not given, the behavior is  
12         unspecified.
- 13      b) Let *i* be the value of the receiver.
- 14      c) If *i* is larger than the value of *num*, return the receiver.
- 15      d) Call *block* with an instance of the class `Integer` whose value is *i*.
- 16      e) Increment *i* by 1 and continue processing from Step c).

## 17   15.2.9   Float

### 18   15.2.9.1   General description

19   Instances of the class `Float` represent floating-point numbers.

20   The precision of the value of an instance of the class `Float` is implementation-defined; however,  
21   if the underlying system of a conforming processor supports IEC 60559, the representation of  
22   an instance of the class `Float` shall be the 64-bit double format as specified in IEC 60559, 3.2.2.

23   When an arithmetic operation involving floating-point numbers results in a value which cannot  
24   be represented exactly as an instance of the class `Float`, the result is rounded to the nearest  
25   representable value. If the two nearest representable values are equally near, which is chosen is  
26   implementation-defined.

27   If the underlying system of a conforming processor supports IEC 60559:

- 28   • If an arithmetic operation involving floating-point numbers results in NaN while invoking  
29    a method of the class `Float`, the behavior of the method is unspecified.

1 Instances of the class `Float` shall not be created by the method `new` of the class `Float`. There-  
2 fore, the singleton method `new` of the class `Float` shall be undefined, by invoking the method  
3 `undef_method` (see 15.2.2.4.42) on the singleton class of the class `Float` with a direct instance  
4 of the class `Symbol` whose name is “new” as the argument.

### 5 **15.2.9.2 Direct superclass**

6 The class `Numeric`

### 7 **15.2.9.3 Instance methods**

#### 8 **15.2.9.3.1 `Float#+`**

---

9 `+(other)`

---

10 **Visibility:** public

11 **Behavior:**

12 a) If *other* is an instance of the class `Float`, return a direct instance of the class `Float`  
13 whose value is the sum of the values of the receiver and *other*.

14 b) If *other* is an instance of the class `Integer`, let *R* be the value of *other* as a floating-  
15 point number.

16 Return a direct instance of the class `Float` whose value is the sum of *R* and the value  
17 of the receiver.

18 c) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument.  
19 Let *V* be the resulting value.

20 1) If *V* is an instance of the class `Array` which contains two elements, let *F* and *S*  
21 be the first and the second element of *V* respectively.

22 i) Invoke the method `+` on *F* with *S* as the only argument.

23 ii) Return the resulting value.

24 2) Otherwise, raise a direct instance of the class `TypeError`.

#### 25 **15.2.9.3.2 `Float#-`**

---

26 `-(other)`

---

27 **Visibility:** public

28 **Behavior:**

29 a) If *other* is an instance of the class `Float`, return a direct instance of the class `Float`  
30 whose value is the result of subtracting the value of *other* from the value of the receiver.

1       b) If *other* is an instance of the class `Integer`, let *R* be the value of *other* as a floating-  
2       point number.

3       Return a direct instance of the class `Float` whose value is the result of subtracting *R*  
4       from the value of the receiver.

5       c) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument.  
6       Let *V* be the resulting value.

7           1) If *V* is an instance of the class `Array` which contains two elements, let *F* and *S*  
8           be the first and the second element of *V* respectively.

9               i) Invoke the method `-` on *F* with *S* as the only argument.

10              ii) Return the resulting value.

11           2) Otherwise, raise a direct instance of the class `TypeError`.

### 12 **15.2.9.3.3 Float#\***

---

13       \*(*other*)

---

14       **Visibility:** public

15       **Behavior:**

16       a) If *other* is an instance of the class `Float`, return a direct instance of the class `Float`  
17       whose value is the result of multiplication of the values of the receiver and *other*.

18       b) If *other* is an instance of the class `Integer`, let *R* be the value of *other* as a floating-  
19       point number.

20       Return a direct instance of the class `Float` whose value is the result of multiplication  
21       of *R* and the value of the receiver.

22       c) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument.  
23       Let *V* be the resulting value.

24           1) If *V* is an instance of the class `Array` which contains two elements, let *F* and *S*  
25           be the first and the second element of *V* respectively.

26               i) Invoke the method `*` on *F* with *S* as the only argument.

27              ii) Return the resulting value.

28           2) Otherwise, raise a direct instance of the class `TypeError`.

### 29 **15.2.9.3.4 Float# /**

---

1 /(*other*)

---

2 **Visibility:** public

3 **Behavior:**

4 a) If *other* is an instance of the class `Float`, return a direct instance of the class `Float`  
5 whose value is the value of the receiver divided by the value of *other*.

6 b) If *other* is an instance of the class `Integer`, let *R* be the value of *other* as a floating-  
7 point number.

8 Return a direct instance of the class `Float` whose value is the value of the receiver  
9 divided by *R*.

10 c) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument.  
11 Let *V* be the resulting value.

12 1) If *V* is an instance of the class `Array` which contains two elements, let *F* and *S*  
13 be the first and the second element of *V* respectively.

14 i) Invoke the method `/` on *F* with *S* as the only argument.

15 ii) Return the resulting value.

16 2) Otherwise, raise a direct instance of the class `TypeError`.

17 **15.2.9.3.5 Float#%**

---

18 %(*other*)

---

19 **Visibility:** public

20 **Behavior:** In the following steps, binary operators `+`, `-`, and `*` represent floating-point  
21 arithmetic operations addition, subtraction, and multiplication which are used in the in-  
22 stance methods `+`, `-`, and `*` of the class `Float`, respectively. The operator `*` has a higher  
23 precedence than the operators `+` and `-`.

24 a) If *other* is an instance of the class `Integer` or the class `Float`:

25 Let *x* be the value of the receiver.

26 1) If *other* is an instance of the class `Float`, let *y* be the value of *other*. If *other* is  
27 an instance of the class `Integer`, let *y* be the value of *other* as a floating-point  
28 number.

29 i) Let *t* be the largest integer smaller than or equal to *x* divided by *y*.

30 ii) Let *m* be  $x - t * y$ .

- 1           iii) If  $m * y < 0$ , return a direct instance of the class `Float` whose value is  $m +$   
2             $y$ .
- 3           iv) Otherwise, return a direct instance of the class `Float` whose value is  $m$ .
- 4        b) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument.  
5        Let  $V$  be the resulting value.
- 6           1) If  $V$  is an instance of the class `Array` which contains two elements, let  $F$  and  $S$   
7            be the first and the second element of  $V$  respectively.
- 8            i) Invoke the method `%` on  $F$  with  $S$  as the only argument.
- 9            ii) Return the resulting value.
- 10          2) Otherwise, raise a direct instance of the class `TypeError`.

### 11 **15.2.9.3.6** `Float#<=>`

---

12        `<=>(other)`

---

13        **Visibility:** public

14        **Behavior:**

- 15        a) If *other* is an instance of the class `Integer` or the class `Float`:
- 16           1) Let  $a$  be the value of the receiver. If *other* is an instance of the class `Float`, let  
17             $b$  be the value of *other*. Otherwise, let  $b$  be the value of *other* as a floating-point  
18            number.
- 19            2) If a conforming processor supports IEC 60559, and if  $a$  or  $b$  is NaN, then return  
20            an implementation-defined value.
- 21            3) If  $a > b$ , return an instance of the class `Integer` whose value is 1.
- 22            4) If  $a = b$ , return an instance of the class `Integer` whose value is 0.
- 23            5) If  $a < b$ , return an instance of the class `Integer` whose value is  $-1$ .
- 24        b) Otherwise, invoke the method `coerce` on *other* with the receiver as the only argument.  
25        Let  $V$  be the resulting value.
- 26           1) If  $V$  is an instance of the class `Array` which contains two elements, let  $F$  and  $S$   
27            be the first and the second element of  $V$  respectively.
- 28            i) Invoke the method `<=>` on  $F$  with  $S$  as the only argument.
- 29            ii) If this invocation does not result in an instance of the class `Integer`, the  
30            behavior is unspecified.

1                   iii) Otherwise, return the value of this invocation.

2                   2) Otherwise, return **nil**.

### 3 **15.2.9.3.7 Float#==**

---

4                   ==( *other* )

---

5                   **Visibility:** public

6                   **Behavior:**

7                   a) If *other* is an instance of the class **Float**:

8                    1) If a conforming processor supports IEC 60559, and if the value of the receiver is  
9                    NaN, then return **false**.

10                   2) If the values of the receiver and *other* are the same number, return **true**.

11                   3) Otherwise, return **false**.

12                   b) If *other* is an instance of the class **Integer**:

13                    1) If the values of the receiver and *other* are the mathematically the same, return  
14                    **true**.

15                    2) Otherwise, return **false**.

16                   c) Otherwise, invoke the method **==** on *other* with the receiver as the argument and return  
17                    the resulting value of this invocation.

### 18 **15.2.9.3.8 Float#ceil**

---

19                   **ceil**

---

20                   **Visibility:** public

21                   **Behavior:** The method returns an instance of the class **Integer** whose value is the smallest  
22                    integer larger than or equal to the value of the receiver.

### 23 **15.2.9.3.9 Float#finite?**

---

24                   **finite?**

---

25                   **Visibility:** public

26                   **Behavior:**

- 1 a) If the value of the receiver is a finite number, return **true**.  
2 b) Otherwise, return **false**.

### 3 **15.2.9.3.10 Float#floor**

---

4 **floor**

---

5 **Visibility:** public

6 **Behavior:** The method returns an instance of the class **Integer** whose value is the largest  
7 integer smaller than or equal to the value of the receiver.

### 8 **15.2.9.3.11 Float#infinite?**

---

9 **infinite?**

---

10 **Visibility:** public

11 **Behavior:**

- 12 a) If the value of the receiver is the positive infinite, return an instance of the class **Integer**  
13 whose value is 1.  
14 b) If the value of the receiver is the negative infinite, return an instance of the class  
15 **Integer** whose value is  $-1$ .  
16 c) Otherwise, return **nil**.

### 17 **15.2.9.3.12 Float#round**

---

18 **round**

---

19 **Visibility:** public

20 **Behavior:** The method returns an instance of the class **Integer** whose value is the nearest  
21 integer to the value of the receiver. If there are two integers equally distant from the value  
22 of the receiver, the one which has the larger absolute value is chosen.

### 23 **15.2.9.3.13 Float#to\_f**

---

24 **to\_f**

---

25 **Visibility:** public

26 **Behavior:** The method returns the receiver.

### 1 15.2.9.3.14 Float#to\_i

---

2 to\_i

---

3 **Visibility:** public

4 **Behavior:** The method returns an instance of the class `Integer` whose value is the integer  
5 part of the receiver.

### 6 15.2.9.3.15 Float#truncate

---

7 truncate

---

8 **Visibility:** public

9 **Behavior:** Same as the method `to_i` (see 15.2.9.3.14).

## 10 15.2.10 String

### 11 15.2.10.1 General description

12 Instances of the class `String` represent sequences of characters. The sequence of characters  
13 represented by an instance of the class `String` is called the **content** of that instance.

14 An instance of the class `String` which does not contain any character is said to be **empty**. An  
15 instance of the class `String` shall be empty when it is created by Step b) of the method `new` of  
16 the class `Class`.

17 The notation “an instance of the class `Object` which represents the character *C*” means either  
18 of the following:

- 19 • An instance of the class `Integer` whose value is the character code of *C*.
- 20 • An instance of the class `String` whose content is the single character *C*.

21 A conforming processor shall choose one of the above representations and use the same repre-  
22 sentation wherever this notation is used.

23 Characters of an instance of the class `String` have their indices counted up from 0. The notation  
24 “the *n*th character of an instance of the class `String`” means the character of the instance whose  
25 index is *n*.

### 26 15.2.10.2 Direct superclass

27 The class `Object`

### 28 15.2.10.3 Included modules

29 The following modules are included in the class `String`.

- 30 • `Comparable`

1 **15.2.10.4 Upper-case and lower-case characters**

2 Some methods of the class `String` handle upper-case and lower-case characters. The correspon-  
3 dence between upper-case and lower-case characters is given in Table 3.

**Table 3 – The correspondence between upper-case and lower-case characters**

upper-case characters	lower-case characters
A	a
B	b
C	c
D	d
E	e
F	f
G	g
H	h
I	i
J	j
K	k
L	l
M	m
N	n
O	o
P	p
Q	q
R	r
S	s
T	t
U	u
V	v
W	w
X	x
Y	y
Z	z

4 **15.2.10.5 Instance methods**

5 **15.2.10.5.1 `String#*`**

---

6 `*(num)`

---

7 **Visibility:** public

1     **Behavior:**

- 2     a) If *num* is not an instance of the class `Integer`, the behavior is unspecified.
- 3     b) Let *n* be the value of the *num*.
- 4     c) If *n* is smaller than 0, raise a direct instance of the class `ArgumentError`.
- 5     d) Otherwise, let *C* be the content of the receiver.
- 6     e) Create a direct instance *S* of the class `String` the content of which is *C* repeated *n*  
7         times.
- 8     f) Return *S*.

9     **15.2.10.5.2 String#+**

---

10     +(*other*)

---

11     **Visibility:** public

12     **Behavior:**

- 13     a) If *other* is not an instance of the class `String`, the behavior is unspecified.
- 14     b) Let *S* and *O* be the contents of the receiver and the *other* respectively.
- 15     c) Return a new direct instance of the class `String` the content of which is the concate-  
16         nation of *S* and *O*.

17     **15.2.10.5.3 String#<=>**

---

18     <=>( *other* )

---

19     **Visibility:** public

20     **Behavior:**

- 21     a) If *other* is not an instance of the class `String`, the behavior is unspecified.
- 22     b) Let *S1* and *S2* be the contents of the receiver and the *other* respectively.
- 23     c) If both *S1* and *S2* are empty, return an instance of the class `Integer` whose value is 0.
- 24     d) Otherwise, if *S1* is empty, return an instance of the class `Integer` whose value is -1.
- 25     e) Otherwise, if *S2* is empty, return an instance of the class `Integer` whose value is 1.
- 26     f) Let *a*, *b* be the character codes of the first characters of *S1* and *S2* respectively.

- 1) If  $a > b$ , return an instance of the class `Integer` whose value is 1.
- 2) If  $a < b$ , return an instance of the class `Integer` whose value is  $-1$ .
- 3) Otherwise, let new  $S1$  and  $S2$  be  $S1$  and  $S2$  excluding their first characters, respectively. Continue processing from Step c).

#### 15.2.10.5.4 `String#==`

---

`==(other)`

---

**Visibility:** public

**Behavior:**

- a) If *other* is not an instance of the class `String`, the behavior is unspecified.
- b) If *other* is an instance of the class `String`:
  - 1) If the contents of the receiver and *other* are the same, return **true**.
  - 2) Otherwise, return **false**.

#### 15.2.10.5.5 `String#=~`

---

`=~(regexp)`

---

**Visibility:** public

**Behavior:**

- a) If *regexp* is not an instance of the class `Regexp`, the behavior is unspecified.
- b) Otherwise, invoke the method `match` on *regexp* with the receiver as the argument (see 15.2.15.7.7), and return the resulting value.

#### 15.2.10.5.6 `String#[]`

---

`[](*args)`

---

**Visibility:** public

**Behavior:**

- a) If the length of *args* is 0 or larger than 2, raise a direct instance of the class `ArgumentError`.
- b) Let  $P$  be the first element of *args*. Let  $n$  be the length of the receiver.

- 1 c) If  $P$  is an instance of the class `Integer`, let  $b$  be the value of  $P$ .
- 2 1) If the length of  $args$  is 1:
- 3 i) If  $b$  is smaller than 0, increment  $b$  by  $n$ . If  $b$  is still smaller than 0, return **nil**.
- 4 ii) If  $b \geq n$ , return **nil**.
- 5 iii) Create an instance of the class `Object` which represents the  $b$ th character of
- 6 the receiver and return this instance.
- 7 2) If the length of  $args$  is 2:
- 8 i) If the last element of  $args$  is an instance of the class `Integer`, let  $l$  be the
- 9 value of the instance. Otherwise, the behavior is unspecified.
- 10 ii) If  $l$  is smaller than 0, or  $b$  is larger than  $n$ , return **nil**.
- 11 iii) If  $b$  is smaller than 0, increment  $b$  by  $n$ . If  $b$  is still smaller than 0, return **nil**.
- 12 iv) If  $b + l$  is larger than  $n$ , let  $l$  be  $n - b$ .
- 13 v) If  $l$  is smaller than or equal to 0, create an empty direct instance of the class
- 14 `String` and return the instance.
- 15 vi) Otherwise, create a direct instance of the class `String` whose content is the
- 16  $(n-l)$  characters of the receiver, from the  $b$ th index, preserving their order.
- 17 Return the instance.
- 18 d) If  $P$  is an instance of the class `Regexp`:
- 19 1) If the length of  $args$  is 1, let  $i$  be 0.
- 20 2) If the length of  $args$  is 2, and the last element of  $args$  is an instance of the class
- 21 `Integer`, let  $i$  be the value of the instance. Otherwise, the behavior is unspecified.
- 22 3) Test if the pattern of  $P$  matches the content of the receiver. (see 15.2.15.4 and
- 23 15.2.15.5). Let  $M$  be the result of the matching process.
- 24 4) If  $M$  is **nil**, return **nil**.
- 25 5) If  $i$  is larger than the length of the match result attribute of  $M$ , return **nil**.
- 26 6) If  $i$  is smaller than 0, increment  $i$  by the length of the match result attribute of
- 27  $M$ . If  $i$  is still smaller than or equal to 0, return **nil**.
- 28 7) Let  $m$  be the  $i$ th element of the match result attribute of  $M$ . Create a direct
- 29 instance of the class `String` whose content is the substring of  $m$  and return the
- 30 instance.
- 31 e) If  $P$  is an instance of the class `String`:

- 1) If the length of *args* is 2, the behavior is unspecified.
  - 2) If the receiver includes the content of *P* as a substring, create a direct instance of the class **String** whose content is equal to the content of *P* and return the instance.
  - 3) Otherwise, return **nil**.
- f) Otherwise, the behavior is unspecified.

#### 15.2.10.5.7 **String#capitalize**

---

capitalize

---

**Visibility:** public

**Behavior:** The method returns a new direct instance of the class **String** which contains all the characters of the receiver, except:

- If the first character of the receiver is a lower-case character, the first character of the resulting instance is the corresponding upper-case character.
- If the *i*th character of the receiver (where *i* > 0) is an upper case character, the *i*th character of the resulting instance is the corresponding lower-case character.

#### 15.2.10.5.8 **String#capitalize!**

---

capitalize!

---

**Visibility:** public

**Behavior:**

- a) Let *s* be the content of the instance of the class **String** returned when the method **capitalize** is invoked on the receiver.
- b) If the content of the receiver and *s* are the same, return **nil**. Otherwise, change the content of the receiver to *s*, and return the receiver.

#### 15.2.10.5.9 **String#chomp**

---

chomp( *rs* = "\n" )

---

**Visibility:** public

**Behavior:**

- 1 a) If *rs* is **nil**, return a new direct instance of the class **String** whose content is the same  
2 as the receiver.
- 3 b) If the receiver is empty, return a new empty direct instance of the class **String**.
- 4 c) If *rs* is not an instance of the class **String**, the behavior is unspecified.
- 5 d) Otherwise, return a new direct instance of the class **String** whose content is the same  
6 as the receiver, except the following characters:
- 7 1) If *rs* consists of only one character 0x0a, the *line-terminator* on the end, if any, is  
8 excluded.
- 9 2) If *rs* is empty, a sequence of *line-terminators* on the end, if any, is excluded.
- 10 3) Otherwise, if the receiver ends with the content of *rs*, this sequence of characters  
11 at the end of the receiver is excluded.

#### 12 **15.2.10.5.10 String#chomp!**

---

13 `chomp!( rs="\n" )`

---

14 **Visibility:** public

15 **Behavior:**

- 16 a) Let *s* be the content of the instance of the class **String** returned when the method  
17 **chomp** is invoked on the receiver with *rs* as the argument.
- 18 b) If the content of the receiver and *s* are the same, return **nil**. Otherwise, change the  
19 content of the receiver to *s*, and return the receiver.

#### 20 **15.2.10.5.11 String#chop**

---

21 `chop`

---

22 **Visibility:** public

23 **Behavior:**

- 24 a) If the receiver is empty, return a new empty direct instance of the class **String**.
- 25 b) Otherwise, create a new direct instance of the class **String** whose content is the receiver  
26 without the last character and return this instance. If the last character is 0x0a, and  
27 the character just before the 0x0a is 0x0d, the 0x0d is also dropped.

#### 28 **15.2.10.5.12 String#chop!**

---

1 chop!

---

2 **Visibility:** public

3 **Behavior:**

4 a) Let  $s$  be the content of the instance of the class **String** returned when the method  
5 chop is invoked on the receiver.

6 b) If the content of the receiver and  $s$  are the same, return **nil**. Otherwise, change the  
7 content of the receiver to  $s$ , and return the receiver.

8 **15.2.10.5.13 String#downcase**

---

9 downcase

---

10 **Visibility:** public

11 **Behavior:** The method returns a new direct instance of the class **String** which contains  
12 all the characters of the receiver, with the upper-case characters replaced with the corre-  
13 sponding lower-case characters.

14 **15.2.10.5.14 String#downcase!**

---

15 downcase!

---

16 **Visibility:** public

17 **Behavior:**

18 a) Let  $s$  be the content of the instance of the class **String** returned when the method  
19 downcase is invoked on the receiver.

20 b) If the content of the receiver and  $s$  are the same, return **nil**. Otherwise, change the  
21 content of the receiver to  $s$ , and return the receiver.

22 **15.2.10.5.15 String#each\_line**

---

23 each\_line(&block)

---

24 **Visibility:** public

25 **Behavior:** Let  $s$  be the content of the receiver. Let  $c$  be the first character of  $s$ .

26 a) If  $block$  is not given, the behavior is unspecified.

- 1       b) Find the first 0x0a in *s* from *c*. If there is such a 0x0a:
- 2           1) Let *d* be that 0x0a.
- 3           2) Create a direct instance *S* of the class **String** whose content is a sequence of  
4           characters from *c* to *d*.
- 5           3) Call *block* with *S* as the argument.
- 6           4) If *d* is the last character of *s*, return the receiver. Otherwise, let new *c* be the  
7           character just after *d* and continue processing from Step b).
- 8       c) If there is not such a 0x0a, create a direct instance of the class **String** whose content is  
9       a sequence of characters from *c* to the last character of *s*. Call *block* with this instance  
10       as the argument.
- 11       d) Return the receiver.

#### 12 **15.2.10.5.16 String#empty?**

---

13       empty?

---

14       **Visibility:** public

15       **Behavior:**

- 16       a) If the receiver is empty, return **true**.
- 17       b) Otherwise, return **false**.

#### 18 **15.2.10.5.17 String#eql?**

---

19       eql?(*other*)

---

20       **Visibility:** public

21       **Behavior:**

- 22       a) If *other* is an instance of the class **String**:
- 23           1) If the contents of the receiver and *other* are the same, return **true**.
- 24           2) Otherwise, return **false**.
- 25       b) If *other* is not an instance of the class **String**, return **false**.

#### 26 **15.2.10.5.18 String#gsub**

---

1 `gsub(*args, &block)`

---

2 **Visibility:** public

3 **Behavior:**

4 a) If the length of *args* is 0 or larger than 2, or the length of *args* is 1 and *block* is not  
5 given, raise a direct instance of the class `ArgumentError`.

6 b) Let *P* be the first element of *args*. If *P* is not an instance of the class `Regexp`, or the  
7 length of *args* is 2 and the last element of *args* is not an instance of the class `String`,  
8 the behavior is unspecified.

9 c) Let *S* be the content of the receiver, and let *l* be the length of *S*.

10 d) Let *L* be an empty list and let *n* be an integer 0.

11 e) Test if the pattern of *P* matches *S* from the index *n* (see 15.2.15.4 and 15.2.15.5). Let  
12 *M* be the result of the matching process.

13 f) If *M* is `nil`, append to *L* the substring of *S* beginning at the *n*th character up to the  
14 last character of *S*.

15 g) Otherwise:

16 1) If the length of *args* is 1:

17 i) Call *block* with a new direct instance of the class `String` whose content is the  
18 matched substring of *M* as the argument.

19 ii) Let *V* be the resulting value of this call. If *V* is not an instance of the class  
20 `String`, the behavior is unspecified.

21 2) Let *pre* be the pre-match (see 15.2.16.1) of *M*. Append to *L* the substring of *pre*  
22 beginning at the *n*th character up to the last character of *pre*, unless *n* is larger  
23 than the index of the last character of *pre*.

24 3) If the length of *args* is 1, append the content of *V* to *L*. If the length of *args* is 2,  
25 append to *L* the content of the last element of *args*.

26 4) Let *post* be the post-match (see 15.2.16.1) of *M*. Let *i* be the index of the first  
27 character of *post* within *S*.

28 i) If *i* is equal to *n*, i.e. if *P* matched an empty string:

29 I) Append to *L* a new direct instance of the class `String` whose content is  
30 the *i*th character of *S*.

31 II) Increment *n* by 1.

- 1                   ii) Otherwise, let new  $n$  be  $i$ .
- 2                   5) If  $n < l$ , continue processing from Step e).
- 3                   h) Create a direct instance of the class **String** whose content is the concatenation of all  
4                   the elements of  $L$ , and return the instance.

5 **15.2.10.5.19 String#gsub!**

---

6                   gsub!(\*args, &block)

---

7                   **Visibility:** public

8                   **Behavior:**

- 9                   a) Let  $s$  be the content of the instance of the class **String** returned when the method  
10                   gsub is invoked on the receiver with the same arguments.
- 11                   b) If the content of the receiver and  $s$  are the same, return **nil**. Otherwise, change the  
12                   content of the receiver to  $s$ , and return the receiver.

13 **15.2.10.5.20 String#hash**

---

14                   hash

---

15                   **Visibility:** public

16                   **Behavior:** The method returns an implementation-defined instance of the class **Integer**  
17                   which satisfies the following condition:

- 18                   a) Let  $S_1$  and  $S_2$  be two distinct instances of the class **String**.
- 19                   b) Let  $H_1$  and  $H_2$  be the resulting values of the invocations of the method **hash** on  $S_1$  and  
20                    $S_2$  respectively.
- 21                   c) If  $S_1$  and  $S_2$  has the same content, the values of  $H_1$  and  $H_2$  shall be the same integer.

22 **15.2.10.5.21 String#include?**

---

23                   include?(obj)

---

24                   **Visibility:** public

25                   **Behavior:**

- 26                   a) If  $obj$  is an instance of the class **Integer**:

- 1           If the receiver includes the character whose character code is the value of *obj*, return  
2           **true**. Otherwise, return **false**.
- 3       b) If *obj* is an instance of the class **String**:
- 4           If there exists a substring of the receiver whose sequence of characters is the same as  
5           the content of *obj*, return **true**. Otherwise, return **false**.
- 6       c) Otherwise, the behavior is unspecified.

#### 7 **15.2.10.5.22 String#index**

---

8       `index( substring, offset=0 )`

---

9       **Visibility:** public

10      **Behavior:**

- 11      a) If *substring* is not an instance of the class **String**, the behavior is unspecified.
- 12      b) Let *R* and *S* be the contents of the receiver and *substring*, respectively.
- 13      c) If *offset* is not an instance of the class **Integer**, the behavior is unspecified.
- 14      d) Let *n* be the value of *offset*.
- 15      e) If *n* is larger than or equal to 0, let *O* be *n*.
- 16      f) Otherwise, let *O* be *l* + *n*, where *l* is the length of *S*.
- 17      g) If *O* is smaller than 0, return **nil**.
- 18      h) If *S* appears as a substring of *R* at one or more positions whose index is larger than  
19          or equal to *O*, return an instance of the class **Integer** whose value is the index of the  
20          first such position.
- 21      i) Otherwise, return **nil**.

#### 22 **15.2.10.5.23 String#initialize**

---

23      `initialize( str="" )`

---

24      **Visibility:** private

25      **Behavior:**

- 26      a) If *str* is not an instance of the class **String**, the behavior is unspecified.
- 27      b) Otherwise, initialize the content of the receiver to the same sequence of characters as  
28          the content of *str*.

1 c) Return an implementation-defined value.

#### 2 15.2.10.5.24 String#initialize\_copy

---

3 initialize\_copy(*original*)

---

4 **Visibility:** private

5 **Behavior:**

6 a) If *original* is not an instance of the class **String**, the behavior is unspecified.

7 b) If *original* is an instance of the class **String**, change the content of the receiver to the  
8 content of *original*.

9 c) Return an implementation-defined value.

#### 10 15.2.10.5.25 String#intern

---

11 intern

---

12 **Visibility:** public

13 **Behavior:**

14 a) If the length of the receiver is 0, or if the receiver contains 0x00, then the behavior is  
15 unspecified.

16 b) Otherwise, return a direct instance of the class **Symbol** whose name is the content of  
17 the receiver.

#### 18 15.2.10.5.26 String#length

---

19 length

---

20 **Visibility:** public

21 **Behavior:** The method returns an instance of the class **Integer** whose value is the number  
22 of characters of the content of the receiver.

#### 23 15.2.10.5.27 String#match

---

24 match(*regexp*)

---

25 **Visibility:** public

1     **Behavior:**

- 2     a) If *regexp* is an instance of the class **Regexp**, let *R* be *regexp*.
- 3     b) Otherwise, if *regexp* is an instance of the class **String**, create a direct instance of  
4       the class **Regexp** by invoking the method **new** on the class **Regexp** with *regexp* as the  
5       argument. Let *R* be the instance of the class **Regexp**.
- 6     c) Otherwise, the behavior is unspecified.
- 7     d) Invoke the method **match** on *R* with the receiver as the argument.
- 8     e) Return the resulting value of the invocation.

9     **15.2.10.5.28 String#replace**

---

10     **replace**( *other* )

---

11     **Visibility:** public

12     **Behavior:** Same as the method **initialize\_copy** (see 15.2.10.5.24).

13     **15.2.10.5.29 String#reverse**

---

14     **reverse**

---

15     **Visibility:** public

16     **Behavior:** The method returns a new direct instance of the class **String** which contains  
17     all the characters of the content of the receiver in the reverse order.

18     **15.2.10.5.30 String#reverse!**

---

19     **reverse!**

---

20     **Visibility:** public

21     **Behavior:**

- 22     a) Change the content of the receiver to the content of the resulting instance of the class  
23       **String** when the method **reverse** is invoked on the receiver.
- 24     b) Return the receiver.

25     **15.2.10.5.31 String#rindex**

---

1 `rindex( substring, offset=nil )`

---

2 **Visibility:** public

3 **Behavior:**

- 4 a) If *substring* is not an instance of the class **String**, the behavior is unspecified.
- 5 b) Let *R* and *S* be the contents of the receiver and *substring*, respectively.
- 6 c) If *offset* is given:
- 7 1) If *offset* is not an instance of the class **Integer**, the behavior is unspecified.
- 8 2) Let *n* be the value of *offset*.
- 9 3) If *n* is larger than or equal to 0, let *O* be *n*.
- 10 4) Otherwise, let *O* be *l* + *n*, where *l* is the length of *S*.
- 11 5) If *O* is smaller than 0, return **nil**.
- 12 d) Otherwise, let *O* be 0.
- 13 e) If *S* appears as a substring of *R* at one or more positions whose index is smaller than
- 14 or equal to *O*, return an instance of the class **Integer** whose value is the index of the
- 15 last such position.
- 16 f) Otherwise, return **nil**.

17 **15.2.10.5.32 String#scan**

---

18 `scan( reg, &block )`

---

19 **Visibility:** public

20 **Behavior:**

- 21 a) If *reg* is not an instance of the class **Regexp**, the behavior is unspecified.
- 22 b) If *block* is not given, create an empty direct instance *A* of the class **Array**.
- 23 c) Let *S* be the content of the receiver, and let *l* be the length of *S*.
- 24 d) Let *n* be an integer 0.
- 25 e) Test if the pattern of *reg* matches *S* from the index *n* (see 15.2.15.4 and 15.2.15.5). Let
- 26 *M* be the result attribute of the matching process.

- 1 f) If  $M$  is not **nil**:
- 2 1) Let  $L$  be the match result attribute of  $M$ .
- 3 2) If the length of  $L$  is 1, create a direct instance  $V$  of the class **String** whose content  
4 is the matched substring of  $M$ .
- 5 3) If the length of  $L$  is larger than 1:
- 6 i) Create an empty direct instance  $V$  of the class **Array**.
- 7 ii) Except for the first element, for each element  $e$  of  $L$ , in the same order in the  
8 list, append to  $V$  a new direct instance of the class **String** whose content is  
9 the substring of  $e$ .
- 10 4) If  $block$  is given, call  $block$  with  $V$  as the argument. Otherwise, append  $V$  to  $A$ .
- 11 5) Let  $post$  be the post-match of  $M$ . Let  $i$  be the index of the first character of  $post$   
12 within  $S$ .
- 13 i) If  $i$  and  $n$  are the same, i.e. if  $reg$  matches the empty string, increment  $n$  by  
14 1.
- 15 ii) Otherwise, let new  $n$  be  $i$ .
- 16 6) If  $n < l$ , continue processing from Step e).
- 17 g) If  $block$  is given, return the receiver. Otherwise, return  $A$ .

### 18 15.2.10.5.33 **String#size**

---

19 **size**

---

20 **Visibility:** public

21 **Behavior:** Same as the method **length** (see 15.2.10.5.26).

### 22 15.2.10.5.34 **String#slice**

---

23 **slice(\*args)**

---

24 **Visibility:** public

25 **Behavior:** Same as the method **[]** (see 15.2.10.5.6).

### 26 15.2.10.5.35 **String#split**

---

1     `split(sep)`

---

2     **Visibility:** public

3     **Behavior:**

- 4     a) If *sep* is not an instance of the class `Regexp`, the behavior is unspecified.
- 5     b) Create an empty direct instance *A* of the class `Array`.
- 6     c) Let *S* be the content of the receiver, and let *l* be the length of *S*.
- 7     d) Let both *sp* and *bp* be 0, and let *was-empty* be false.
- 8     e) Test if the pattern of *sep* matches *S* from the index *sp* (see 15.2.15.4 and 15.2.15.5).  
9     Let *M* be the result of the matching process.
- 10    f) If *M* is **nil**, append to *A* a new direct instance of the class `String` whose content is the  
11    substring of *S* beginning at the *sp*th character up to the last character of *S*.
- 12    g) Otherwise:
- 13       1) If the matched substring of *M* is an empty string:
- 14            i) If *was-empty* is true, append to *A* a new direct instance of the class `String`  
15            whose content is the *bp*th character of *S*.
- 16            ii) Otherwise, increment *sp* by 1. If *sp* < *l*, let new *was-empty* be true and  
17            continue processing from Step e).
- 18       2) Otherwise, let new *was-empty* be false. Let *pre* be the pre-match of *M*. Append  
19       to *A* a new direct instance of the class `String` whose content is the substring of  
20       *pre* beginning at the *bp*th character up to the last character of *pre*, unless *bp* is  
21       larger than the index of the last character of *pre*.
- 22       3) Let *L* be the match result attribute of *M*.
- 23       4) If the length of *L* is larger than 1, except for the first element, for each element *e*  
24       of *L*, in the same order in the list, take the following steps:
- 25            i) Let *c* be the substring of *e*.
- 26            ii) If *c* is not **nil**, append to *A* a new direct instance of the class `String` whose  
27            content is *c*.
- 28       5) Let *post* be the post-match of *M*, and replace both *sp* and *bp* with the index of  
29       the first character of *post*.
- 30       6) If *sp* > *l*, continue processing from Step e).

- 1 h) If the last element of *A* is an instance of the class **String** whose content is empty,  
2 remove the element. Repeat this step until this condition does not hold.
- 3 i) Return *A*.

#### 4 **15.2.10.5.36 String#sub**

---

5 `sub(*args, &block)`

---

6 **Visibility:** public

7 **Behavior:**

- 8 a) If the length of *args* is 1 and *block* is given, or the length of *args* is 2:
- 9 1) If the first element of *args* is not an instance of the class **Regexp**, the behavior is  
10 unspecified.
  - 11 2) Test if the pattern of the first element of *args* matches the content of the receiver  
12 (see 15.2.15.4 and 15.2.15.5). Let *M* be the result of the matching process.
  - 13 3) If *M* is **nil**, create a direct instance of the class **String** whose content is the same  
14 as the receiver and return the instance.
  - 15 4) Otherwise:
    - 16 i) If the length of *args* is 1, call *block* with a new direct instance of the class  
17 **String** whose content is the matched substring of *M* as the argument. Let *S*  
18 be the resulting value of this call. If *S* is not an instance of the class **String**,  
19 the behavior is unspecified.
    - 20 ii) If the length of *args* is 2, let *S* be the last element of *args*. If *S* is not an  
21 instance of the class **String**, the behavior is unspecified.
    - 22 iii) Create a direct instance of the class **String** whose content is the concatenation  
23 of pre-match of *M*, the content of *S*, and post-match of *M*, and return the  
24 instance.
- 25 b) Otherwise, raise a direct instance of the class **ArgumentError**.

#### 26 **15.2.10.5.37 String#sub!**

---

27 `sub!(*args, &block)`

---

28 **Visibility:** public

29 **Behavior:**

- 1 a) Let  $s$  be the content of the instance of the class `String` returned when the method `sub`  
2 is invoked on the receiver with the same arguments.
- 3 b) If the content of the receiver and  $s$  are the same, return `nil`. Otherwise, change the  
4 content of the receiver to  $s$ , and return the receiver.

5 **15.2.10.5.38 String#to\_i**

---

6 `to_i( base=10 )`

---

7 **Visibility:** public

8 **Behavior:**

- 9 a) If  $base$  is not an instance of the class `Integer` whose value is 2, 8, 10, nor 16, the  
10 behavior is unspecified. Otherwise, let  $b$  be the value of  $base$ .
- 11 b) If the receiver is empty, return an instance of the class `Integer` whose value is 0.
- 12 c) Let  $i$  be 0. Increment  $i$  by 1 while the  $i$ th character of the receiver is a *whitespace*  
13 character.
- 14 d) If the  $i$ th character of the receiver is “+” or “-”, increment  $i$  by 1.
- 15 e) If the  $i$ th character of the receiver is “0”, and any of the following conditions holds,  
16 increment  $i$  by 2:
- 17 Let  $c$  be the character of the receiver whose index is  $i$  plus 1.
- 18 •  $b$  is 2, and  $c$  is “b” or “B”.
  - 19 •  $b$  is 8, and  $c$  is “o” or “O”.
  - 20 •  $b$  is 10, and  $c$  is “d” or “D”.
  - 21 •  $b$  is 16, and  $c$  is “x” or “X”.
- 22 f) Let  $s$  be a sequence of the following characters of the receiver from the  $i$ th index:
- 23 • If  $b$  is 2, *binary-digit* and “\_”.
  - 24 • If  $b$  is 8, *octal-digit* and “\_”.
  - 25 • If  $b$  is 10, *decimal-digit* and “\_”.
  - 26 • If  $b$  is 16, *hexadecimal-digit* and “\_”.
- 27 g) If the length of  $s$  is 0, return an instance of the class `Integer` whose value is 0.
- 28 h) If  $s$  starts with “\_”, or  $s$  contains successive “\_”s, the behavior is unspecified.

- 1 i) Let  $n$  be the value of  $s$ , ignoring interleaving “\_”s, computed in base  $b$ .  
2 If the “-” occurs in Step d), return an instance of the class `Integer` whose value is  
3  $-n$ . Otherwise, return an instance of the class `Integer` whose value is  $n$ .

#### 4 **15.2.10.5.39 String#to\_f**

---

5 `to_f`

---

6 **Visibility:** public

7 **Behavior:**

- 8 a) If the receiver is empty, return a direct instance of the class `Float` whose value is 0.0.  
9 b) If the receiver starts with a sequence of characters which is a *float-literal*, return a direct  
10 instance of the class `Float` whose value is the value of the *float-literal* (see 8.7.6.2).  
11 c) If the receiver starts with a sequence of characters which is a *unprefixed-decimal-integer-*  
12 *literal*, return a direct instance of the class `Float` whose value is the value of the  
13 *unprefixed-decimal-integer-literal* as a floating-point number (see 8.7.6.2).  
14 d) Otherwise, return a direct instance of the class `Float` whose value is implementation-  
15 defined.

#### 16 **15.2.10.5.40 String#to\_s**

---

17 `to_s`

---

18 **Visibility:** public

19 **Behavior:**

- 20 a) If the receiver is a direct instance of the class `String`, return the receiver.  
21 b) Otherwise, create a new direct instance of the class `String` whose content is the same  
22 as the content of the receiver and return this instance.

#### 23 **15.2.10.5.41 String#to\_sym**

---

24 `to_sym`

---

25 **Visibility:** public

26 **Behavior:** Same as the method `intern` (see 15.2.10.5.25).

#### 27 **15.2.10.5.42 String#upcase**

---

1     **upcase**

---

2     **Visibility:** public

3     **Behavior:** The method returns a new direct instance of the class **String** which contains  
4     all the characters of the receiver, with all the lower-case characters replaced with the cor-  
5     responding upper-case characters.

6     **15.2.10.5.43 String#upcase!**

---

7     **upcase!**

---

8     **Visibility:** public

9     **Behavior:**

- 10    a) Let *s* be the content of the instance of the class **String** returned when the method  
11    **upcase** is invoked on the receiver.
- 12    b) If the content of the receiver and *s* are the same, return **nil**. Otherwise, change the  
13    content of the receiver to *s*, and return the receiver.

14    **15.2.11 Symbol**

15    **15.2.11.1 General description**

16    Instances of the class **Symbol** represent names (see 8.7.6.6). No two instances of the class **Symbol**  
17    shall represent the same name.

18    Instances of the class **Symbol** shall not be created by the method **new** of the class **Symbol**.  
19    Therefore, the singleton method **new** of the class **Symbol** shall be undefined, by invoking the  
20    method **undef\_method** (see 15.2.2.4.42) on the singleton class of the class **Symbol** with a direct  
21    instance of the class **Symbol** whose name is “new” as the argument.

22    **15.2.11.2 Direct superclass**

23    The class **Object**

24    **15.2.11.3 Instance methods**

25    **15.2.11.3.1 Symbol#===**

---

26    **===(other)**

---

27    **Visibility:** public

28    **Behavior:** Same as the method **==** of the module **Kernel** (see 15.3.1.3.1).

### 1 15.2.11.3.2 Symbol#id2name

---

2 id2name

---

3 **Visibility:** public

4 **Behavior:** The method creates a direct instance of the class **String**, the content of which  
5 represents the name of the receiver, and returns this instance.

### 6 15.2.11.3.3 Symbol#to\_s

---

7 to\_s

---

8 **Visibility:** public

9 **Behavior:** Same as the method `id2name` (see 15.2.11.3.2).

### 10 15.2.11.3.4 Symbol#to\_sym

---

11 to\_sym

---

12 **Visibility:** public

13 **Behavior:** The method returns the receiver.

## 14 15.2.12 Array

### 15 15.2.12.1 General description

16 Instances of the class **Array** represent arrays, which are unbounded. An instance of the class  
17 **Array** which has no element is said to be **empty**. The number of elements in an instance of the  
18 class **Array** is called its **length**.

19 Instances of the class **Array** shall be empty when they are created by Step b) of the method **new**  
20 of the class **Class**.

21 Elements of an instance of the class **Array** have their indices counted up from 0.

22 Given an instance *A* of the class **Array**, operations **append**, **prepend**, and **remove** are defined  
23 as follows:

24 **append:** To append an object *O* to *A* is defined as follows:

25 Insert *O* after the last element of *A*.

26 Appending an object to *A* increases its length by 1.

1     **prepend:** To prepend an object  $O$  to  $A$  is defined as follows:

2     Insert  $O$  to the first index of  $A$ . Original elements of  $A$  are moved toward the end of  $A$  by  
3     one position.

4     Prepending an object to  $A$  increases its length by 1.

5     **remove:** To remove an element  $X$  from  $A$  is defined as follows:

6     a) Remove  $X$  from  $A$ .

7     b) If  $X$  is not the last element of  $A$ , move the elements after  $X$  toward the head of  $A$  by  
8     one position.

9     Removing an object to  $A$  decreases its length by 1.

#### 10   **15.2.12.2 Direct superclass**

11   The class `Object`

#### 12   **15.2.12.3 Included modules**

13   The following module is included in the class `Array`.

- 14   • `Enumerable`

#### 15   **15.2.12.4 Singleton methods**

##### 16   **15.2.12.4.1 `Array.[]`**

---

17   `Array. [] (*items)`

---

18   **Visibility:** public

19   **Behavior:** The method returns a newly created instance of the class `Array` which contains  
20   the elements of *items*, preserving their order.

#### 21   **15.2.12.5 Instance methods**

##### 22   **15.2.12.5.1 `Array#*`**

---

23   `*( num )`

---

24   **Visibility:** public

25   **Behavior:**

26   a) If *num* is not an instance of the class `Integer`, the behavior is unspecified.

- 1 b) If the value of *num* is smaller than 0, raise a direct instance of the class `ArgumentError`.
- 2 c) If the value of *num* is 0, return an empty direct instance of the class `Array`.
- 3 d) Otherwise, create an empty direct instance *A* of the class `Array` and repeat the following
- 4 for *num* times:
- 5 • Append all the elements of the receiver to *A*, preserving their order.
- 6 e) Return *A*.

#### 7 **15.2.12.5.2** `Array#+`

---

8 `+(other)`

---

9 **Visibility:** public

10 **Behavior:**

- 11 a) If *other* is an instance of the class `Array`, let *A* be *other*. Otherwise, the behavior is
- 12 unspecified.
- 13 b) Create an empty direct instance *R* of the class `Array`.
- 14 c) For each element of the receiver, in the indexing order, append the element to *R*. Then,
- 15 for each element of *A*, in the indexing order, append the element to *R*.
- 16 d) Return *R*.

#### 17 **15.2.12.5.3** `Array#<<`

---

18 `<<(obj)`

---

19 **Visibility:** public

20 **Behavior:** The method appends *obj* to the receiver and return the receiver.

#### 21 **15.2.12.5.4** `Array#[[]]`

---

22 `[](*args)`

---

23 **Visibility:** public

24 **Behavior:**

- 25 a) Let *n* be the length of the receiver.

- 1       b) If the length of *args* is 0, raise a direct instance of the class `ArgumentError`.
- 2       c) If the length of *args* is 1:
- 3           1) If the only argument is an instance of the class `Integer`, let *k* be the value of the
- 4           only argument. Otherwise, the behavior is unspecified.
- 5           2) If  $k < 0$ , increment *k* by *n*. If *k* is still smaller than 0, return `nil`.
- 6           3) If  $k \geq n$ , return `nil`.
- 7           4) Otherwise, return the *k*th element of the receiver.
- 8       d) If the length of *args* is 2:
- 9           1) If the elements of *args* are instances of the class `Integer`, let *b* and *l* be the values
- 10           of the first and the last element of *args*, respectively. Otherwise, the behavior is
- 11           unspecified.
- 12           2) If  $b < 0$ , increment *b* by *n*. If *b* is still smaller than 0, return `nil`.
- 13           3) If  $b > n$  or  $l < 0$ , return `nil`.
- 14           4) If  $b = n$ , create an empty direct instance of the class `Array` and return this instance.
- 15           5) If  $l > n - b$ , let new *l* be  $n - b$ .
- 16           6) Create an empty direct instance *A* of the class `Array`. Append the *l* elements of
- 17           the receiver to *A*, from the *b*th index, preserving their order. Return *A*.
- 18       e) If the length of *args* is larger than 2, raise a direct instance of the class `ArgumentError`.

19 **15.2.12.5.5** `Array#[]=`

---

20 `[] = (*args)`

---

21 **Visibility:** public

22 **Behavior:**

- 23       a) Let *n* be the length of the receiver.
- 24       b) If the length of *args* is smaller than 2, raise a direct instance of the class `ArgumentError`.
- 25       c) If the length of *args* is 2:
- 26           1) If the first element of *args* is an instance of the class `Integer`, let *k* be the value
- 27           of the element and let *V* be the last element of *args*. Otherwise, the behavior is
- 28           unspecified.

- 1           2) If  $k < 0$ , increment  $k$  by  $n$ . If  $k$  is still smaller than 0, raise a direct instance of  
2           the class `IndexError`.
- 3           3) If  $k < n$ , replace the  $k$ th element of the receiver with  $V$ .
- 4           4) Otherwise, expand the length of the receiver to  $k + 1$ . The last element of the  
5           receiver is  $V$ . If  $k > n$ , the elements whose index is from  $n$  to  $k - 1$  is **nil**.
- 6           5) Return  $V$ .
- 7           d) If the length of *args* is 3, the behavior is unspecified.
- 8           e) If the length of *args* is larger than 3, raise a direct instance of the class `ArgumentError`.

#### 9 **15.2.12.5.6** `Array#clear`

---

10 `clear`

---

11 **Visibility:** public

12 **Behavior:** The method removes all the elements from the receiver and return the receiver.

#### 13 **15.2.12.5.7** `Array#collect!`

---

14 `collect!(&block)`

---

15 **Visibility:** public

16 **Behavior:**

17 a) If *block* is given:

18           1) For each element of the receiver in the indexing order, call *block* with the element  
19           as the only argument and replace the element with the resulting value.

20           2) Return the receiver.

21 b) If *block* is not given, the behavior is unspecified.

#### 22 **15.2.12.5.8** `Array#concat`

---

23 `concat(other)`

---

24 **Visibility:** public

25 **Behavior:**

- 1 a) If *other* is not an instance of the class **Array**, the behavior is unspecified.
- 2 b) Otherwise, append all the elements of *other* to the receiver, preserving their order.
- 3 c) Return the receiver.

#### 4 **15.2.12.5.9 Array#delete\_at**

---

5 `delete_at(index)`

---

6 **Visibility:** public

7 **Behavior:**

- 8 a) If the *index* is not an instance of the class **Integer**, the behavior is unspecified.
- 9 b) Otherwise, let *i* be the value of the *index*.
- 10 c) Let *n* be the length of the receiver.
- 11 d) If *i* is smaller than 0, increment *i* by *n*. If *i* is still smaller than 0, return **nil**.
- 12 e) If *i* is larger than or equal to *n*, return **nil**.
- 13 f) Otherwise, remove the *i*th element of the receiver, and return the removed element.

#### 14 **15.2.12.5.10 Array#each**

---

15 `each(&block)`

---

16 **Visibility:** public

17 **Behavior:**

- 18 a) If *block* is given:
  - 19 1) For each element of the receiver in the indexing order, call *block* with the element
  - 20 as the only argument.
  - 21 2) Return the receiver.
- 22 b) If *block* is not given, the behavior is unspecified.

#### 23 **15.2.12.5.11 Array#each\_index**

---

1 `each_index(&block)`

---

2 **Visibility:** public

3 **Behavior:**

4 a) If *block* is given:

5 1) For each element of the receiver in the indexing order, call *block* with an argument,  
6 which is an instance of the class `Integer` whose value is the index of the element.

7 2) Return the receiver.

8 b) If *block* is not given, the behavior is unspecified.

9 **15.2.12.5.12 Array#empty?**

---

10 `empty?`

---

11 **Visibility:** public

12 **Behavior:**

13 a) If the receiver is empty, return **true**.

14 b) Otherwise, return **false**.

15 **15.2.12.5.13 Array#first**

---

16 `first(*args)`

---

17 **Visibility:** public

18 **Behavior:**

19 a) If the length of *args* is 0:

20 1) If the receiver is empty, return **nil**.

21 2) Otherwise, return the first element of the receiver.

22 b) If the length of *args* is 1:

23 1) If the only argument is not an instance of the class `Integer`, the behavior is  
24 unspecified. Otherwise, let *n* be the value of the only argument.

- 1           2) If  $n$  is smaller than 0, raise a direct instance of the class `ArgumentError`.
- 2           3) Otherwise, let  $N$  be the smaller of  $n$  and the length of the receiver.
- 3           4) Return a newly created instance of the class `Array` which contains the first  $N$
- 4           elements of the receiver, preserving their order.
- 5           c) If the length of *args* is larger than 1, raise a direct instance of the class `ArgumentError`.

#### 6 **15.2.12.5.14** `Array#index`

---

7 `index( object=nil )`

---

8 **Visibility:** public

9 **Behavior:**

- 10 a) If *object* is given:
  - 11 1) For each element  $E$  of the receiver in the indexing order, take the following steps:
    - 12 i) Invoke the method `==` on  $E$  with *object* as the argument.
    - 13 ii) If the resulting value is a trueish object, return an instance of the class
    - 14 `Integer` whose value is the index of  $E$ .
  - 15 2) If an instance of the class `Integer` is not returned in Step a) 1) ii), return **nil**.
- 16 b) Otherwise, the behavior is unspecified.

#### 17 **15.2.12.5.15** `Array#initialize`

---

18 `initialize( size=0, obj=nil, &block )`

---

19 **Visibility:** private

20 **Behavior:**

- 21 a) If *size* is not an instance of the class `Integer`, the behavior is unspecified. Otherwise,
- 22 let  $n$  be the value of *size*.
- 23 b) If  $n$  is smaller than 0, raise a direct instance of the class `ArgumentError`.
- 24 c) Remove all the elements from the receiver.
- 25 d) If  $n$  is 0, return an implementation-defined value.
- 26 e) If  $n$  is larger than 0:

- 1) If *block* is given:
  - i) Let *k* be 0.
  - ii) Call *block* with an argument, which is an instance of the class **Integer** whose value is *k*. Append the resulting value of this call to the receiver.
  - iii) Increase *k* by 1. If *k* is equal to *n*, terminate this process. Otherwise, repeat from Step e) 1) ii).
- 2) Otherwise, append *obj* to the receiver *n* times.
- 3) Return an implementation-defined value.

#### 15.2.12.5.16 **Array#initialize\_copy**

---

`initialize_copy(original)`

---

**Visibility:** private

**Behavior:**

- a) If *original* is not an instance of the class **Array**, the behavior is unspecified.
- b) Remove all the elements from the receiver.
- c) Append all the elements of *original* to the receiver, preserving their order.
- d) Return an implementation-defined value.

#### 15.2.12.5.17 **Array#join**

---

`join(sep=nil)`

---

**Visibility:** public

**Behavior:**

- a) If *sep* is neither **nil** nor an instance of the class **String**, the behavior is unspecified.
- b) Create an empty direct instance *S* of the class **String**.
- c) For each element *X* of the receiver, in the indexing order:
  - 1) If *sep* is not **nil**, and *X* is not the first element of the receiver, append the content of *sep* to *S*.
  - 2) If *X* is an instance of the class **String**, append the content of *X* to *S*.

- 1           3) If  $X$  is an instance of the class **Array**:
- 2            i) If  $X$  is the receiver, i.e. if the receiver contains itself, append an implementation-  
3            defined sequence of characters to  $S$ .
- 4            ii) Otherwise, append to  $S$  the content of the instance of the class **String** re-  
5            turned by the invocation of the method `join` on  $X$  with  $sep$  as the argument.
- 6           4) Otherwise, the behavior is unspecified.
- 7        d) Return  $S$ .

8 **15.2.12.5.18 Array#last**

---

9        `last(*args)`

---

10       **Visibility:** public

11       **Behavior:**

12       a) If the length of  $args$  is 0:

13           1) If the receiver is empty, return **nil**.

14           2) Otherwise, return the last element of the receiver.

15       b) If the length of  $args$  is 1:

16           1) If the only argument is not an instance of the class **Integer**, the behavior is  
17           unspecified. Otherwise, let  $n$  be the value of the only argument.

18           2) If  $n$  is smaller than 0, raise a direct instance of the class **ArgumentError**.

19           3) Otherwise, let  $N$  be the smaller of  $n$  and the length of the receiver.

20           Return a newly created instance of the class **Array** which contains the last  $N$   
21           elements of the receiver, preserving their order.

22       c) If the length of  $args$  is larger than 1, raise a direct instance of the class **ArgumentError**.

23 **15.2.12.5.19 Array#length**

---

24       `length`

---

25       **Visibility:** public

26       **Behavior:** The method returns an instance of the class **Integer** whose value is the number  
27       of elements of the receiver.

1 **15.2.12.5.20 Array#map!**

---

2 `map!(&block)`

---

3 **Visibility:** public

4 **Behavior:** Same as the method `collect!` (see 15.2.12.5.7).

5 **15.2.12.5.21 Array#pop**

---

6 `pop`

---

7 **Visibility:** public

8 **Behavior:**

9 a) If the receiver is empty, return **nil**.

10 b) Otherwise, remove the last element from the receiver and return that element.

11 **15.2.12.5.22 Array#push**

---

12 `push(*items)`

---

13 **Visibility:** public

14 **Behavior:**

15 a) For each element of *items*, in the indexing order, append it to the receiver.

16 b) Return the receiver.

17 **15.2.12.5.23 Array#replace**

---

18 `replace(other)`

---

19 **Visibility:** public

20 **Behavior:** Same as the method `initialize_copy` (see 15.2.12.5.16).

21 **15.2.12.5.24 Array#reverse**

---

1     **reverse**

---

2     **Visibility:** public

3     **Behavior:** The method returns a newly created instance of the class **Array** which contains  
4     all the elements of the receiver in the reverse order.

5     **15.2.12.5.25   Array#reverse!**

---

6     **reverse!**

---

7     **Visibility:** public

8     **Behavior:** The method reverses the order of the elements of the receiver and return the  
9     receiver.

10    **15.2.12.5.26   Array#rindex**

---

11    **rindex( *object*=nil )**

---

12    **Visibility:** public

13    **Behavior:**

14    a) If *object* is given:

15        1) For each element *E* of the receiver in the reverse indexing order, take the following  
16        steps:

17            i) Invoke the method `==` on *E* with *object* as the argument.

18            ii) If the resulting value is a trueish object, return an instance of the class  
19            **Integer** whose value is the index of *E*.

20        2) If an instance of the class **Integer** is not returned in Step a) 1) ii), return **nil**.

21    b) Otherwise, the behavior is unspecified.

22    **15.2.12.5.27   Array#shift**

---

23    **shift**

---

24    **Visibility:** public

25    **Behavior:**

- 1 a) If the receiver is empty, return **nil**.  
2 b) Otherwise, remove the first element from the receiver and return that element.

### 3 **15.2.12.5.28 Array#size**

---

4 **size**

---

5 **Visibility:** public

6 **Behavior:** Same as the method `length` (see 15.2.12.5.19).

### 7 **15.2.12.5.29 Array#slice**

---

8 **slice(\*args)**

---

9 **Visibility:** public

10 **Behavior:** Same as the method `[]` (see 15.2.12.5.4).

### 11 **15.2.12.5.30 Array#unshift**

---

12 **unshift(\*items)**

---

13 **Visibility:** public

14 **Behavior:**

- 15 a) For each element of *items*, in the reverse indexing order, prepend it to the receiver.  
16 b) Return the receiver.

## 17 **15.2.13 Hash**

### 18 **15.2.13.1 General description**

19 Instances of the class `Hash` represent hashes, which are sets of key/value pairs.

20 An instance of the class `Hash` which has no key/value pair is said to be **empty**. Instances of  
21 the class `Hash` shall be empty when they are created by Step b) of the method `new` of the class  
22 `Class`.

23 An instance of the class `Hash` cannot contain more than one key/value pair for each key.

24 An instance of the class `Hash` has the following attribute:

25 **default value or proc:** Either of the followings:

1 • A default value, which is returned by the method [] when the specified key is not  
2 found in the instance of the class `Hash`.

3 • A default proc, which is an instance of the class `Proc` and used to generate the return  
4 value of the method [] when the specified key is not found in the instance of the class  
5 `Hash`.

6 An instance of the class `Hash` shall not have both a default value and a default proc simul-  
7 taneously.

8 Given two keys  $K_1$  and  $K_2$ , the notation “ $K_1 \equiv K_2$ ” means that the keys are equivalent, i.e. all  
9 of the following conditions hold:

10 • An invocation of the method `eq1?` on  $K_1$  with  $K_2$  as the only argument evaluates to a  
11 trueish object.

12 • Let  $H_1$  and  $H_2$  be the results of invocations of the method `hash` on  $K_1$  and  $K_2$ , respectively.

13  $H_1$  and  $H_2$  are the instances of the class `Integer` which represents the same integer.

14 A conforming processor may define a certain range of integers, and when the values of  $H_1$   
15 or  $H_2$  lies outside of this range, the processor may convert  $H_1$  or  $H_2$  to another instance of  
16 the class `Integer` whose value is within the range. Let  $I_1$  and  $I_2$  be each of the resulting  
17 instances respectively.

18 The values of  $I_1$  and  $I_2$  are the same integer.

19 If  $H_1$  or  $H_2$  is not an instance of the class `Integer`, whether  $K_1 \equiv K_2$  is unspecified.

20 NOTE  $K_1 \equiv K_2$  is not equivalent to  $K_2 \equiv K_1$ .

### 21 15.2.13.2 Direct superclass

22 The class `Object`

### 23 15.2.13.3 Included modules

24 The following module is included in the class `Hash`.

25 • `Enumerable`

### 26 15.2.13.4 Instance methods

#### 27 15.2.13.4.1 `Hash#==`

---

28 `==(other)`

---

29 **Visibility:** public

30 **Behavior:**

- 1 a) If *other* is not an instance of the class **Hash**, the behavior is unspecified.
- 2 b) If all of the following conditions hold, return **true**:
- 3     • The receiver and *other* have the same number of key/value pairs.
- 4     • For each key/value pair *P* in the receiver, *other* has a corresponding key/value
- 5         pair *Q* which satisfies the following conditions:
- 6             — The key of *P*  $\equiv$  the key of *Q*.
- 7             — An invocation of the method **==** on the value of *P* with the value of *Q* as an
- 8                 argument results in a trueish object.
- 9 c) Otherwise, return **false**.

#### 10 15.2.13.4.2 Hash#[*key*]

---

11 [*key*]

---

12 **Visibility:** public

13 **Behavior:**

- 14 a) If the receiver has a key/value pair *P* where *key*  $\equiv$  the key of *P*, return the value of *P*.
- 15 b) Otherwise, invoke the method **default** on the receiver with *key* as the argument and
- 16     return the resulting value.

#### 17 15.2.13.4.3 Hash#[*key*]=*value*]

---

18 [*key*]=(*key*, *value*)

---

19 **Visibility:** public

20 **Behavior:**

- 21 a) If the receiver has a key/value pair *P* where *key*  $\equiv$  the key of *P*, replace the value of *P*
- 22     with *value*.
- 23 b) Otherwise:
- 24     1) If *key* is a direct instance of the class **String**, create a copy of *key*, i.e. create a
- 25         direct instance *K* of the class **String** whose content is the same as the *key*.
- 26     2) If *key* is not an instance of the class **String**, let *K* be *key*.
- 27     3) If *key* is an instance of a subclass of the class **String**, whether to create a copy or
- 28         not is implementation-defined.

- 1           4) Store a pair of *K* and *value* into the receiver.
- 2           c) Return *value*.

3 **15.2.13.4.4 Hash#clear**

---

4           **clear**

---

5           **Visibility:** public

6           **Behavior:**

- 7           a) Remove all the key/value pairs from the receiver.
- 8           b) Return the receiver.

9 **15.2.13.4.5 Hash#default**

---

10          **default(\*args)**

---

11          **Visibility:** public

12          **Behavior:**

- 13          a) If the length of *args* is larger than 1, raise a direct instance of the class **ArgumentError**.
- 14          b) If the receiver has the default value, return the value.
- 15          c) If the receiver has the default proc:
- 16             1) If the length of *args* is 0, return **nil**.
- 17             2) If the length of *args* is 1, invoke the method **call** on the default proc of the
- 18                 receiver with two arguments, the receiver and the only element of *args*. Return
- 19                 the resulting value of this invocation.
- 20          d) Otherwise, return **nil**.

21 **15.2.13.4.6 Hash#default=**

---

22          **default=(value)**

---

23          **Visibility:** public

24          **Behavior:**

- 25          a) If the receiver has the default proc, remove the default proc.

- 1        b) Set the default value of the receiver to *value*.
- 2        c) Return *value*.

#### 3    **15.2.13.4.7 Hash#default\_proc**

---

4        `default_proc`

---

5        **Visibility:** public

6        **Behavior:**

- 7        a) If the receiver has the default proc, return the default proc.
- 8        b) Otherwise, return **nil**.

#### 9    **15.2.13.4.8 Hash#delete**

---

10       `delete(key, &block)`

---

11       **Visibility:** public

12       **Behavior:**

- 13       a) If the receiver has a key/value pair *P* where *key*  $\equiv$  the key of *P*, remove *P* from the  
14       receiver and return the value of *P*.
- 15       b) Otherwise:
  - 16           1) If *block* is given, call *block* with *key* as the argument. Return the resulting value  
17           of this call.
  - 18           2) Otherwise, return **nil**.

#### 19   **15.2.13.4.9 Hash#each**

---

20       `each(&block)`

---

21       **Visibility:** public

22       **Behavior:**

- 23       a) If *block* is given, for each key/value pair of the receiver in an implementation-defined  
24       order:
  - 25           1) Create a direct instance of the class **Array** which contains two elements, the key  
26           and the value of the pair.

1           2) Call *block* with the instance as an argument.

2           Return the receiver.

3           b) If *block* is not given, the behavior is unspecified.

#### 4 **15.2.13.4.10 Hash#each\_key**

---

5           `each_key(&block)`

---

6           **Visibility:** public

7           **Behavior:**

8           a) If *block* is given, for each key/value pair of the receiver, in an implementation-defined  
9           order, call *block* with the key of the pair as the argument. Return the receiver.

10          b) If *block* is not given, the behavior is unspecified.

#### 11 **15.2.13.4.11 Hash#each\_value**

---

12          `each_value(&block)`

---

13          **Visibility:** public

14          **Behavior:**

15          a) If *block* is given, call *block* for each key/value pair of the receiver, with the value as the  
16          argument, in an implementation-defined order. Return the receiver.

17          b) If *block* is not given, the behavior is unspecified.

#### 18 **15.2.13.4.12 Hash#empty?**

---

19          `empty?`

---

20          **Visibility:** public

21          **Behavior:**

22          a) If the receiver is empty, return **true**.

23          b) Otherwise, return **false**.

#### 24 **15.2.13.4.13 Hash#has\_key?**

---

1 `has_key?( key )`

---

2 **Visibility:** public

3 **Behavior:**

4 a) If the receiver has a key/value pair *P* where *key*  $\equiv$  the key of *P*, return **true**.

5 b) Otherwise, return **false**.

6 **15.2.13.4.14 Hash#has\_value?**

---

7 `has_value?( value )`

---

8 **Visibility:** public

9 **Behavior:**

10 a) If the receiver has a key/value pair whose value holds the following condition, return  
11 **true**.

12 • An invocation of the method `==` on the value with *value* as the argument result in  
13 a trueish object.

14 b) Otherwise, return **false**.

15 **15.2.13.4.15 Hash#include?**

---

16 `include?( key )`

---

17 **Visibility:** public

18 **Behavior:** Same as the method `has_key?` (see 15.2.13.4.13).

19 **15.2.13.4.16 Hash#initialize**

---

20 `initialize( *args, &block )`

---

21 **Visibility:** private

22 **Behavior:**

23 a) If *block* is given, and the length of *args* is not 0, raise a direct instance of the class  
24 `ArgumentError`.

- 1 b) If *block* is given and the length of *args* is 0, create a direct instance of the class `Proc`  
2 which represents *block* and set the default proc of the receiver to this instance.
- 3 c) If *block* is not given:
- 4 1) If the length of *args* is 0, let *D* be `nil`.
- 5 2) If the length of *args* is 1, let *D* be the only argument.
- 6 3) If the length of *args* is larger than 1, raise a direct instance of the class `ArgumentError`.
- 7 4) Set the default value of the receiver to *D*.
- 8 d) Return an implementation-defined value.

#### 9 **15.2.13.4.17 Hash#initialize\_copy**

---

10 `initialize_copy(original)`

---

11 **Visibility:** private

12 **Behavior:**

- 13 a) If *original* is not an instance of the class `Hash`, the behavior is unspecified.
- 14 b) Remove all the key/value pairs from the receiver.
- 15 c) For each key/value pair *P* of *original*, in an implementation-defined order, add or  
16 update a key/value pair of the receiver by invoking the method `[]=` (see 15.2.13.4.3)  
17 on the receiver with the key of *P* and the value of *P* as the arguments.
- 18 d) Remove the default value or the default proc from the receiver.
- 19 e) If *original* has a default value, set the default value of the receiver to that value.
- 20 f) If *original* has a default proc, set the default proc of the receiver to that proc.
- 21 g) Return an implementation-defined value.

#### 22 **15.2.13.4.18 Hash#key?**

---

23 `key?(key)`

---

24 **Visibility:** public

25 **Behavior:** Same as the method `has_key?` (see 15.2.13.4.13).

#### 26 **15.2.13.4.19 Hash#keys**

---

1 keys

---

2 **Visibility:** public

3 **Behavior:** The method returns a newly created instance of the class `Array` whose content  
4 is all the keys of the receiver. The order of the keys stored is implementation-defined.

5 **15.2.13.4.20 Hash#length**

---

6 length

---

7 **Visibility:** public

8 **Behavior:** The method returns an instance of the class `Integer` whose value is the number  
9 of key/value pairs stored in the receiver.

10 **15.2.13.4.21 Hash#member?**

---

11 member?( *key* )

---

12 **Visibility:** public

13 **Behavior:** Same as the method `has_key?` (see 15.2.13.4.13).

14 **15.2.13.4.22 Hash#merge**

---

15 merge( *other*, &*block* )

---

16 **Visibility:** public

17 **Behavior:**

- 18 a) If *other* is not an instance of the class `Hash`, the behavior is unspecified.
- 19 b) Otherwise, create a direct instance *H* of the class `Hash` which has the same key/value  
20 pairs as the receiver.
- 21 c) For each key/value pair *P* of *other*, in an implementation-defined order:
- 22 1) If *block* is given:
- 23 i) If *H* has the key/value pair *Q* where the key of *P*  $\equiv$  the key of *Q*, call *block*  
24 with three arguments, the key of *P*, the value of *Q*, and the value of *P*. Let  
25 *V* be the resulting value. Add or update a key/value pair of the receiver by  
26 invoking the method `[]=` (see 15.2.13.4.3) on *H* with the key of *P* and *V* as  
27 the arguments.

- 1           ii) Otherwise, add or update a key/value pair of the receiver by invoking the  
2           method []= (see 15.2.13.4.3) on *H* with the key of *P* and the value of *P* as  
3           the arguments.
- 4           2) If *block* is not given, add or update a key/value pair of the receiver by invoking  
5           the method []= (see 15.2.13.4.3) on *H* with the key of *P* and the value of *P* as  
6           the arguments.
- 7           d) Return *H*.

#### 8 **15.2.13.4.23 Hash#replace**

---

9       *replace( other )*

---

10       **Visibility:** public

11       **Behavior:** Same as the method `initialize_copy` (see 15.2.13.4.17).

#### 12 **15.2.13.4.24 Hash#shift**

---

13       *shift*

---

14       **Visibility:** public

15       **Behavior:**

16       a) If the receiver is empty:

17           1) If the receiver has the default proc, invoke the method `call` on the default proc  
18           with two arguments, the receiver and **nil**. Return the resulting value of this call.

19           2) If the receiver has the default value, return the value.

20           3) Otherwise, return **nil**.

21       b) Otherwise, choose a key/value pair *P* and remove *P* from the receiver. Return a newly  
22       created instance of the class `Array` which contains two elements, the key and the value  
23       of *P*.

24       Which pair is chosen is implementation-defined.

#### 25 **15.2.13.4.25 Hash#size**

---

26       *size*

---

27       **Visibility:** public

28       **Behavior:** Same as the method `length` (see 15.2.13.4.20).

#### 1 15.2.13.4.26 Hash#store

---

2 store(*key*, *value*)

---

3 **Visibility:** public

4 **Behavior:** Same as the method []= (see 15.2.13.4.3).

#### 5 15.2.13.4.27 Hash#value?

---

6 value?(*value*)

---

7 **Visibility:** public

8 **Behavior:** Same as the method has\_value? (see 15.2.13.4.14).

#### 9 15.2.13.4.28 Hash#values

---

10 values

---

11 **Visibility:** public

12 **Behavior:** The method returns a newly created instance of the class `Array` which contains  
13 all the values of the receiver. The order of the values stored is implementation-defined.

### 14 15.2.14 Range

#### 15 15.2.14.1 General description

16 Instances of the class `Range` represent ranges between two values, the start and end points.

17 An instance of the class `Range` has the following attributes:

18 **start point:** The value at the start of the range.

19 **end point:** The value at the end of the range.

20 **exclusive flag:** If this is true, the end point is excluded from the range. Otherwise, the  
21 end point is included in the range.

22 When the method `clone` (see 15.3.1.3.8) or the method `dup` (see 15.3.1.3.9) of the class `Kernel`  
23 is invoked on an instance of the class `Range`, those attributes shall be copied from the receiver  
24 to the resulting value.

#### 25 15.2.14.2 Direct superclass

26 The class `Object`

1 **15.2.14.3 Included modules**

2 The following module is included in the class `Range`.

- 3 • `Enumerable`

4 **15.2.14.4 Instance methods**

5 **15.2.14.4.1 `Range#==`**

---

6 `==(other)`

---

7 **Visibility:** public

8 **Behavior:**

9 a) If all of the following conditions hold, return **true**:

- 10 • *other* is an instance of the class `Range`.
- 11 • Let *S* be the start point of *other*. Invocation of the method `==` on the start point  
12 of the receiver with *S* as the argument results in a trueish object.
- 13 • Let *E* be the end point of *other*. Invocation of the method `==` on the end point of  
14 the receiver with *E* as the argument results in a trueish object.
- 15 • The exclusive flags of the receiver and *other* are the same boolean value.

16 b) Otherwise, return **false**.

17 **15.2.14.4.2 `Range#===`**

---

18 `===(obj)`

---

19 **Visibility:** public

20 **Behavior:**

21 a) If neither the start point of the receiver nor the end point of the receiver is an instance  
22 of the class `Numeric`, the behavior is unspecified.

23 b) Invoke the method `<=>` on the start point of the receiver with *obj* as the argument.  
24 Let *S* be the result of this invocation.

25 1) If *S* is not an instance of the class `Integer`, the behavior is unspecified.

26 2) If the value of *S* is larger than 0, return **false**.

- 1 c) Invoke the method `<=>` on *obj* with the end point of the receiver as the argument. Let  
2 *E* be the result of this invocation.
- 3 • If *E* is not an instance of the class `Integer`, the behavior is unspecified.
  - 4 • If the exclusive flag of the receiver is true, and the value of *E* is smaller than 0,  
5 return **true**.
  - 6 • If the exclusive flag of the receiver is false, and the value of *E* is smaller than or  
7 equal to 0, return **true**.
  - 8 • Otherwise, return **false**.

#### 9 15.2.14.4.3 `Range#begin`

---

10 `begin`

---

11 **Visibility:** public

12 **Behavior:** The method returns the start point of the receiver.

#### 13 15.2.14.4.4 `Range#each`

---

14 `each(&block)`

---

15 **Visibility:** public

16 **Behavior:**

- 17 a) If *block* is not given, the behavior is unspecified.
- 18 b) If an invocation of the method `respond_to?` on the start point of the receiver with a  
19 direct instance of the class `Symbol` whose name is `succ` as the argument results in a  
20 falseish object, raise a direct instance of the class `TypeError`.
- 21 c) Let *V* be the start point of the receiver.
- 22 d) Invoke the method `<=>` on *V* with the end point of the receiver as the argument. Let  
23 *C* be the resulting value.
  - 24 1) If *C* is not an instance of the class `Integer`, the behavior is unspecified.
  - 25 2) If the value of *C* is larger than 0, return the receiver.
  - 26 3) If the value of *C* is 0:
    - 27 i) If the exclusive flag of the receiver is true, return the receiver.
    - 28 ii) If the exclusive flag of the receiver is false, call *block* with *V* as the argument,  
29 then, return the receiver.

- 1 e) Call *block* with *V* as the argument.
- 2 f) Invoke the method `succ` on *V* with no argument, and let new *V* be the resulting value.
- 3 g) Continue processing from Step d).

#### 4 **15.2.14.4.5 Range#end**

---

5 `end`

---

6 **Visibility:** public

7 **Behavior:** The method returns the end point of the receiver.

#### 8 **15.2.14.4.6 Range#exclude\_end?**

---

9 `exclude_end?`

---

10 **Visibility:** public

11 **Behavior:** If the exclusive flag of the receiver is true, return **true**. Otherwise, return **false**.

#### 12 **15.2.14.4.7 Range#first**

---

13 `first`

---

14 **Visibility:** public

15 **Behavior:** Same as the method `begin` (see 15.2.14.4.3).

#### 16 **15.2.14.4.8 Range#include?**

---

17 `include?(obj)`

---

18 **Visibility:** public

19 **Behavior:** Same as the method `===` (see 15.2.14.4.2).

#### 20 **15.2.14.4.9 Range#initialize**

---

21 `initialize(left, right, exclusive=false)`

---

22 **Visibility:** private

1     **Behavior:**

- 2     a)    Invoke the method `<=>` on *left* with *right* as the argument. If an exception is raised and  
3         not handled during this invocation, raise a direct instance of the class `ArgumentError`.  
4         If the result of this invocation is not an instance of the class `Integer`, the behavior is  
5         unspecified.
- 6     b)    If *exclusive* is a trueish object, let *f* be true. Otherwise, let *f* be false.
- 7     c)    Set the start point, end point, and exclusive flag of the receiver to *left*, *right*, and *f*,  
8         respectively.
- 9     d)    Return an implementation-defined value.

10  **15.2.14.4.10 Range#last**

---

11     **last**

---

12     **Visibility:** public

13     **Behavior:** Same as the method `end` (see 15.2.14.4.5).

14  **15.2.14.4.11 Range#member?**

---

15     **member?(*obj*)**

---

16     **Visibility:** public

17     **Behavior:** Same as the method `===` (see 15.2.14.4.2).

18  **15.2.15 Regexp**

19  **15.2.15.1 General description**

20  Instances of the class `Regexp` represent regular expressions, and have the following attributes.

21     **pattern:** A *pattern* of the regular expression (see 15.2.15.4). The default value of this  
22     attribute is empty.

23     If the value of this attribute is empty when a method is invoked on an instance of the class  
24     `Regexp`, except for the invocation of the method `initialize`, the behavior of the invoked  
25     method is unspecified.

26     **ignorecase-flag:** A boolean value which indicates whether a match is performed in the  
27     case insensitive manner. The default value of this attribute is false.

28     **multiline-flag:** A boolean value which indicates whether the pattern “.” matches a *line-*  
29     *terminator* (see 15.2.15.4). The default value of this attribute is false.

## 1 15.2.15.2 Direct superclass

2 The class `Object`

## 3 15.2.15.3 Constants

4 The following constants are defined in the class `Regexp`.

5 **IGNORECASE:** An instance of the class `Integer` whose value is  $2^n$ , where the integer  $n$   
6 is an implementation-defined value. The value of this constant shall be different from that  
7 of `MULTILINE` described below.

8 **MULTILINE:** An instance of the class `Integer` whose value is  $2^m$ , where the integer  $m$   
9 is an implementation-defined value.

10 The above constants are used to set the `ignorecase-flag` and `multiline-flag` attributes of an in-  
11 stance of the class `Regexp` (see 15.2.15.7.1).

## 12 15.2.15.4 Patterns

### 13 Syntax

---

14 *pattern* ::  
15     *alternative*  
16     | *pattern*<sub>1</sub> | *alternative*<sub>2</sub>

17 *alternative* ::  
18     [ empty ]  
19     | *alternative*<sub>3</sub> *term*

20 *term* ::  
21     *anchor*  
22     | *atom*<sub>1</sub>  
23     | *atom*<sub>2</sub> *quantifier*

24 *anchor* ::  
25     *left-anchor* | *right-anchor*

26 *left-anchor* ::  
27     \`\A` | `^`

28 *right-anchor* ::  
29     \`\z` | `$`

30 *quantifier* ::  
31     `*` | `+` | `?`

```

1  atom ::
2      pattern-character
3      | grouping
4      | .
5      | atom-escape-sequence

6  pattern-character ::
7      source-character but not regexp-meta-character

8  regexp-meta-character ::
9      | | . | * | + | ^ | ? | ( | ) | # | \ | $
10     | future-reserved-meta-character

11 future-reserved-meta-character ::
12     [ | ] | { | }

13 grouping ::
14     ( pattern )

15 atom-escape-sequence ::
16     decimal-escape-sequence
17     | regexp-character-escape-sequence

18 decimal-escape-sequence ::
19     \ decimal-digit-except-zero

20 regexp-character-escape-sequence ::
21     regexp-escape-sequence
22     | regexp-non-escaped-sequence
23     | hexadecimal-escape-sequence
24     | regexp-octal-escape-sequence
25     | regexp-control-escape-sequence

26 regexp-escape-sequence ::
27     \ regexp-escaped-character

28 regexp-escaped-character ::
29     n | t | r | f | v | a | e

30 regexp-non-escaped-sequence ::
31     \ regexp-meta-character

32 regexp-octal-escape-sequence ::
33     octal-escape-sequence but not decimal-escape-sequence

```

1 *regexp-control-escape-sequence* ::  
2     \`( C- | c ) regexp-control-escaped-character`

3 *regexp-control-escaped-character* ::  
4     *regexp-character-escape-sequence*  
5     | ?  
6     | *source-character* **but not** ( \`\ | ?` )

---

7 *future-reserved-meta-characters* are reserved for the extension of the pattern of regular expres-  
8 sions.

## 9 Semantics

10 A regular expression selects specific substrings from a string called a target string according  
11 to the *pattern* of the regular expression. If the *pattern* matches more than one substring, the  
12 substring which begins earliest in the target string is selected. If there is more than one such  
13 substring beginning at that point, the substring that has the highest priority, which is described  
14 below, is selected. Each component of the *pattern* matches a substring of the target string as  
15 follows:

16 a) A *pattern* matches the following substring:

- 17 1) If the *pattern* is an *alternative*<sub>1</sub>, it matches the string matched with the *alternative*<sub>1</sub>.
- 18 2) If the *pattern* is a *pattern*<sub>1</sub> | *alternative*<sub>2</sub>, it matches the string matched with either the  
19 *pattern*<sub>1</sub> or the *alternative*<sub>2</sub>. The one matched with the *pattern*<sub>1</sub> has a higher priority.

20 EXAMPLE 1 `"ab".slice(/(a|ab)/)` returns "a", not "ab".

21 b) An *alternative* matches the following substring:

- 22 1) If the *alternative* is [empty], it matches an empty string.
- 23 2) If the *alternative* is an *alternative*<sub>3</sub> *term*, the *alternative* matches the substring whose  
24 first part is matched with the *alternative*<sub>3</sub> and whose rest part is matched with the  
25 *term*.

26 If there is more than one such substring, the priority of the substrings is determined  
27 as follows:

- 28 i) If there is more than one candidate which is matched with the *alternative*<sub>3</sub>, a  
29 substring whose first part is a candidate with a higher priority has a higher priority.

30 EXAMPLE 2 `"abc".slice(/(a|ab)(c|b)/)` returns "ab", not "abc". In this case,  
31 (a|ab) is prior to (c|b).

- 32 ii) If the first parts of substrings are the same, and if there is more than one candidate  
33 which is matched with the *term*, a substring whose rest part is a candidate with a  
34 higher priority has a higher priority.

35 EXAMPLE 3 `"abc".slice(/a(b|bc)/)` returns "ab", not "abc".

1 c) A *term* matches the following substring:

2 1) If the *term* is an *atom*<sub>1</sub>, it matches the string matched with the *atom*<sub>1</sub>.

3 2) If the *term* is an *atom*<sub>2</sub> *quantifier*, it matches a string as follows:

4 i) If the *quantifier* is \*, it matches a sequence of zero or more strings matched with  
5 the *atom*<sub>2</sub>.

6 ii) If the *quantifier* is +, it matches a sequence of one or more strings matched with  
7 *atom*<sub>2</sub>.

8 iii) If the *quantifier* is ?, it matches a sequence of zero or one string matched with the  
9 *atom*<sub>2</sub>.

10 A longer sequence has a higher priority.

11 EXAMPLE 4 `"aaa".slice(/a*/)` returns "aaa", none of "", "a", and "aa".

12 3) If the *term* is an *anchor*, it matches the empty string at a specific position within the  
13 target string *S*, as follows:

14 i) If the *anchor* is \A, it matches an empty string at the beginning of *S*.

15 ii) If the *anchor* is ^, it matches an empty string at the beginning of *S* or just after  
16 a *line-terminator* which is followed by at least one character.

17 iii) If the *anchor* is \z, it matches an empty string at the end of *S*.

18 iv) If the *anchor* is \$, it matches an empty string at the end of *S* or just before a  
19 *line-terminator*.

20 d) An *atom* matches the following substring:

21 1) If the *atom* is a *pattern-character*, it matches a character *C* represented by the *pattern-*  
22 *character*. If the *atom* is present in the pattern of an instance of the class `Regexp` whose  
23 `ignorecase-flag` attribute is true, it also matches a corresponding upper-case character  
24 of *C*, if *C* is a lower-case character, or a corresponding lower-case character of *C*, if *C*  
25 is an upper-case character.

26 2) If the *atom* is a *grouping*, it matches the string matched with the *grouping*.

27 3) If the *atom* is ".", it matches any character except for a *line-terminator*. If the *atom* is  
28 present in the pattern of an instance of the class `Regexp` whose `multiline-flag` attribute  
29 is true, it also matches a *line-terminator*.

30 4) If the *atom* is an *atom-escape-sequence*, it matches the string matched with the *atom-*  
31 *escape-sequence*.

32 e) A *grouping* matches the substring matched with the *pattern*.

33 f) An *atom-escape-sequence* matches the following substring:

- 1) If the *atom-escape-sequence* is a *decimal-escape-sequence*, it matches the string matched with the *decimal-escape-sequence*.
- 2) If the *atom-escape-sequence* is a *regexp-character-escape-sequence*, it matches a string of length one, the content of which is the character represented by the *regexp-character-escape-sequence*.
- g) A *decimal-escape-sequence* matches the following substring:
- 1) Let  $i$  be an integer represented by *decimal-digit-except-zero*.
  - 2) Let  $G$  be the  $i$ th *grouping* in the *pattern*, counted from 1, in the order of the occurrence of “(” of *groupings* from the left of the *pattern*.
  - 3) If the *decimal-escape-sequence* is present before  $G$  within the *pattern*, it does not match any string.
  - 4) If  $G$  matches any string, the *decimal-escape-sequence* matches the same string.
  - 5) Otherwise, the *decimal-escape-sequence* does not match any string.
- h) A *regexp-character-escape-sequence* represents a character as follows:
- A *regexp-escape-sequence* represents a character as shown in 8.7.6.3.3, Table 1.
  - A *regexp-non-escaped-sequence* represents a *regexp-meta-character*.
  - A *hexadecimal-escape-sequence* represents a character as described in 8.7.6.3.3.
  - A *regexp-octal-escape-sequence* is interpreted in the same way as an *octal-escape-sequence* (see 8.7.6.3.3).
  - A *regexp-control-escape-sequence* represents a character, the code of which is computed by taking bitwise AND of 0x9f and the code of the character represented by the *regexp-control-escaped-character*, except when the *regexp-control-escaped-character* is ?, in which case, the *regexp-control-escape-sequence* represents a character whose code is 127.

### 15.2.15.5 Matching process

A *pattern*  $P$  is considered to successfully match the given string  $S$ , if there exists a substring of  $S$  (including  $S$  itself) matched with  $P$ .

- a) When an index is specified, it is tested if  $P$  matches the part of  $S$  which begins at the index and ends at the end of  $S$ . However, if the match succeeds, the string attribute of the resulting instance of the class `MatchData` is  $S$ , not the part of  $S$  which begins at the index, as described below.
- b) A matching process returns either an instance of the class `MatchData` (see 15.2.16) if the match succeeds or `nil` if the match fails.
- c) An instance of the class `MatchData` is created as follows:

- 1) Let  $B$  be the substring of  $S$  which  $P$  matched.
  - 2) Create a direct instance of the class `MatchData`, and let  $M$  be the instance.
  - 3) Set the string attribute of  $M$  (see 15.2.16.1) to  $S$ .
  - 4) Create a new empty list  $L$ .
  - 5) Let  $O$  be a pair of the substring  $B$  and the index of the first character of  $B$  within  $S$ . Append  $O$  to  $L$ .
  - 6) For each *grouping*  $G$  in  $P$ , in the order of the occurrence of its “(” within  $P$ , take the following steps:
    - i) If  $G$  matches a substring of  $B$  under the matching process of  $P$ , let  $B_G$  be the substring. Let  $O$  be a pair of the substring  $B_G$  and the index of the first character of  $B_G$  within  $S$ . Append  $O$  to  $L$ .
    - ii) Otherwise, append to  $L$  a pair whose substring and index of the substring are **nil**.
  - 7) Set the match result attribute of  $M$  to  $L$ .
  - 8)  $M$  is the instance of the class `MatchData` returned by the matching process.
- d) A matching process creates or updates a local variable binding with name “~”, which is specifically used by the method `Regexp.last_match` (see 15.2.15.6.3), as follows:
- 1) Let  $M$  be the value which the matching process returns.
  - 2) If the binding for the name “~” can be resolved by the process described in 9.2 as if “~” were a *local-variable-identifier*, replace the value of the binding with  $M$ .
  - 3) Otherwise, create a local variable binding with name “~” and value  $M$  in the uppermost non-block element of `[[local-variable-bindings]]` where the non-block element means the element which does not correspond to a *block*.
- e) A conforming processor may name the binding other than “~”; however, it shall not be of the form *local-variable-identifier*.

## 15.2.15.6 Singleton methods

### 15.2.15.6.1 `Regexp.compile`

---

```
Regexp.compile(*args)
```

---

**Visibility:** public

**Behavior:** Same as the method `new` (see 15.2.3.3.3).

### 15.2.15.6.2 `Regexp.escape`

---

1 `Regexp.escape(string)`

---

2 **Visibility:** public

3 **Behavior:**

- 4 a) If *string* is not an instance of the class `String`, the behavior is unspecified.
- 5 b) Let *S* be the content of *string*.
- 6 c) Return a new direct instance of the class `String` whose content is the same as *S*,  
7 except that every occurrences of characters on the left of Table 4 are replaced with the  
8 corresponding sequences of characters on the right of Table 4.

**Table 4 – Regexp escaped characters**

Characters replaced	Escaped sequence
0x0a	<code>\n</code>
0x09	<code>\t</code>
0x0d	<code>\r</code>
0x0c	<code>\f</code>
0x20	<code>\ 0x20</code>
<code>#</code>	<code>\#</code>
<code>\$</code>	<code>\\$</code>
<code>(</code>	<code>\(</code>
<code>)</code>	<code>\)</code>
<code>*</code>	<code>\*</code>
<code>+</code>	<code>\+</code>
<code>-</code>	<code>\-</code>
<code>.</code>	<code>\.</code>
<code>?</code>	<code>\?</code>
<code>[</code>	<code>\[</code>
<code>\</code>	<code>\\</code>
<code>]</code>	<code>\]</code>
<code>^</code>	<code>\^</code>
<code>{</code>	<code>\{</code>
<code> </code>	<code>\ </code>
<code>}</code>	<code>\}</code>

9 **15.2.15.6.3 Regexp.last\_match**

---

10 `Regexp.last_match(*index)`

---

1     **Visibility:** public

2     **Behavior:**

- 3     a) Search for a binding of a local variable with name “~” as described in 9.2 as if “~” were  
4        a *local-variable-identifier*.
- 5     b) If the binding is found and its value is an instance of the class `MatchData`, let *M* be  
6        the instance. Otherwise, return **nil**.
- 7     c) If the length of *index* is 0, return *M*.
- 8     d) If the length of *index* is larger than 1, raise a direct instance of the class `ArgumentError`.
- 9     e) If the length of *index* is 1, let *A* be the only argument.
- 10    f) If *A* is not an instance of the class `Integer`, the behavior of the method is unspecified.
- 11    g) Let *R* be the result returned by invoking the method `[]` (see 15.2.16.3.1) on *M* with  
12        *A* as the only argument.
- 13    h) Return *R*.

#### 14 15.2.15.6.4 `Regexp.quote`

---

15     `Regexp.quote`

---

16     **Visibility:** public

17     **Behavior:** Same as the method `escape` (see 15.2.15.6.2).

#### 18 15.2.15.7 Instance methods

##### 19 15.2.15.7.1 `Regexp#initialize`

---

20     `initialize( source, flag=nil )`

---

21     **Visibility:** private

22     **Behavior:**

- 23     a) If *source* is an instance of the class `Regexp`, let *S* be the pattern attribute of *source*.  
24        If *source* is an instance of the class `String`, let *S* be the content of *source*. Otherwise,  
25        the behavior is unspecified.
- 26     b) If *S* is not of the form *pattern* (see 15.2.15.4), raise a direct instance of the class  
27        `RegexpError`.
- 28     c) Set the pattern attribute of the receiver to *S*.

- 1 d) If *flag* is an instance of the class `Integer`, let *n* be the value of the instance.
- 2 1) If computing bitwise AND of the value of the constant `IGNORECASE` of the class  
3 `Regexp` and *n* results in non-zero value, set the ignorecase-flag attribute of the  
4 receiver to true.
- 5 2) If computing bitwise AND of the value of the constant `MULTILINE` of the class  
6 `Regexp` and *n* results in non-zero value, set the multiline-flag attribute of the  
7 receiver to true.
- 8 e) If *flag* is not an instance of the class `Integer`, and if *flag* is a trueish object, then set  
9 the ignorecase-flag attribute of the receiver to true.
- 10 f) Return an implementation-defined value.

### 11 15.2.15.7.2 `Regexp#initialize_copy`

---

12 `initialize_copy( original )`

---

13 **Visibility:** private

14 **Behavior:**

- 15 a) If *original* is not an instance of the class of the receiver, raise a direct instance of the  
16 class `TypeError`.
- 17 b) Set the pattern attribute of the receiver to the pattern attribute of *original*.
- 18 c) Set the ignorecase-flag attribute of the receiver to the ignorecase-flag attribute of *orig-*  
19 *inal*.
- 20 d) Set the multiline-flag attribute of the receiver to the multiline-flag attribute of *original*.
- 21 e) Return an implementation-defined value.

### 22 15.2.15.7.3 `Regexp#===`

---

23 `==( other )`

---

24 **Visibility:** public

25 **Behavior:**

- 26 a) If *other* is not an instance of the class `Regexp`, return **false**.
- 27 b) If the corresponding attributes of the receiver and *other* are the same, return **true**.
- 28 c) Otherwise, return **false**.

1 **15.2.15.7.4** **Regexp#===**

---

2 `===( string )`

---

3 **Visibility:** public

4 **Behavior:**

- 5 a) If *string* is not an instance of the class **String**, the behavior is unspecified.
- 6 b) Let *S* be the content of *string*.
- 7 c) Test if the pattern of the receiver matches *S* (see 15.2.15.4 and 15.2.15.5). Let *M* be  
8 the result of the matching process.
- 9 d) If *M* is an instance of the class **MatchData**, return **true**.
- 10 e) Otherwise, return **false**.

11 **15.2.15.7.5** **Regexp#=~**

---

12 `=~( string )`

---

13 **Visibility:** public

14 **Behavior:**

- 15 a) If *string* is not an instance of the class **String**, the behavior is unspecified.
- 16 b) Let *S* be the content of *string*.
- 17 c) Test if the pattern of the receiver matches *S* (see 15.2.15.4 and 15.2.15.5). Let *M* be  
18 the result of the matching process.
- 19 d) If *M* is **nil** return **nil**.
- 20 e) If *M* is an instance of the class **MatchData**, let *P* be first element of the match result  
21 attribute of *M*, and let *i* be the index of the substring of *P*.
- 22 f) Return an instance of the class **Integer** whose value is *i*.

23 **15.2.15.7.6** **Regexp#casefold?**

---

24 `casefold?`

---

25 **Visibility:** public

26 **Behavior:** The method returns the value of the `ignorecase-flag` attribute of the receiver.

### 1 15.2.15.7.7 `Regexp#match`

---

2 `match( string )`

---

3 **Visibility:** public

4 **Behavior:**

- 5 a) If *string* is not an instance of the class `String`, the behavior is unspecified.
- 6 b) Let *S* be the content of *string*.
- 7 c) Test if the pattern of the receiver matches *S* (see 15.2.15.4 and 15.2.15.5). Let *M* be  
8 the result of the matching process.
- 9 d) Return *M*.

### 10 15.2.15.7.8 `Regexp#source`

---

11 `source`

---

12 **Visibility:** public

13 **Behavior:** The method returns a direct instance of the class `String` whose content is the  
14 pattern of the receiver.

## 15 15.2.16 `MatchData`

### 16 15.2.16.1 General description

17 Instances of the class `MatchData` represent results of successful matches of instances of the class  
18 `Regexp` against instances of the class `String`.

19 An instance of the class `MatchData` has the attributes called *string* and *match result*, which  
20 are initialized as described in 15.2.15.5. The string attribute is the target string *S* of a matching  
21 process. The match result attribute is a list whose element is a pair of a substring *B* matched  
22 by the *pattern* of an instance of the class `Regexp` or a *grouping* in the *pattern*, and the index *I*  
23 of the first character of *B* within *S*. *B* is called the substring of the element, and *I* is called the  
24 index of the substring of the element. Elements of the match result attribute are indexed by  
25 integers starting from 0.

26 Given an instance *M* of the class `MatchData`, three values named *matched substring*, *pre-*  
27 *match* and *post-match* of *M*, respectively, are defined as follows:

28 Let *S* be the string attribute of *M*. Let *F* be the first element of the match result attribute of  
29 *M*. Let *B* and *O* be the substring of *F* and the index of the substring of *F*. Let *i* be the sum of  
30 *O* and the length of *B*.

31 **matched substring:** The matched substring of *M* is *B*.

1     **pre-match:** The pre-match of  $M$  is a part of  $S$ , from the first up to, but not including the  
2      $O$ th character of  $S$ .

3     **post-match:** The post-match of  $M$  is a part of  $S$ , from the  $i$ th up to the last character of  
4      $S$ .

## 5   **15.2.16.2   Direct superclass**

6   The class `Object`

## 7   **15.2.16.3   Instance methods**

### 8   **15.2.16.3.1   MatchData#[]**

---

9     `[] (*args)`

---

10    **Visibility:** public

11    **Behavior:** Invoke the method `to_a` on the receiver (see 15.2.16.3.12), and invoke the  
12    method `[]` on the resulting instance of the class `Array` with `args` as the arguments (see  
13    15.2.12.5.4), and then, return the resulting value of the invocation of the method `[]`.

### 14   **15.2.16.3.2   MatchData#begin**

---

15     `begin(index)`

---

16    **Visibility:** public

17    **Behavior:**

18    a) If `index` is not an instance of the class `Integer`, the behavior is unspecified.

19    b) Let  $L$  be the match result attribute of the receiver, and let  $i$  be the value of `index`.

20    c) If  $i$  is smaller than 0, or larger than or equal to the number of elements of  $L$ , raise a  
21    direct instance of the class `IndexError`.

22    d) Otherwise, return the second portion of the  $i$ th element of  $L$ .

### 23   **15.2.16.3.3   MatchData#captures**

---

24     `captures`

---

25    **Visibility:** public

26    **Behavior:**

- 1 a) Let  $L$  be the match result attribute of the receiver.
- 2 b) Create an empty direct instance  $A$  of the class `Array`.
- 3 c) Except for the first element, for each element  $e$  of  $L$ , in the same order in the list,  
4 append to  $A$  a direct instance of the class `String` whose content is the substring of  $e$ .
- 5 d) Return  $A$ .

#### 6 **15.2.16.3.4 MatchData#end**

---

7 `end(index)`

---

8 **Visibility:** public

9 **Behavior:**

- 10 a) If  $index$  is not an instance of the class `Integer`, the behavior is unspecified.
- 11 b) Let  $L$  be the match result attribute of the receiver, and let  $i$  be the value of  $index$ .
- 12 c) If  $i$  is smaller than 0, or larger than or equal to the number of elements of  $L$ , raise a  
13 direct instance of the class `IndexError`.
- 14 d) Let  $F$  and  $S$  be the substring and the index of the substring of the  $i$ th element of  $L$ ,  
15 respectively.
- 16 e) If  $F$  is `nil`, return `nil`.
- 17 f) Otherwise, let  $f$  be the length of  $F$ . Return an instance of the class `Integer` whose  
18 value is the sum of  $S$  and  $f$ .

#### 19 **15.2.16.3.5 MatchData#initialize\_copy**

---

20 `initialize_copy(original)`

---

21 **Visibility:** private

22 **Behavior:**

- 23 a) If  $original$  is not an instance of the class of the receiver, raise a direct instance of the  
24 class `TypeError`.
- 25 b) Set the string attribute of the receiver to the string attribute of  $original$ .
- 26 c) Set the match result attribute of the receiver to the match result attribute of  $original$ .
- 27 d) Return an implementation-defined value.

### 1 15.2.16.3.6 MatchData#length

---

2 length

---

3 **Visibility:** public

4 **Behavior:**

5 The method returns the number of elements of the match result attribute of the receiver.

### 6 15.2.16.3.7 MatchData#offset

---

7 `offset(index)`

---

8 **Visibility:** public

9 **Behavior:**

- 10 a) If *index* is not an instance of the class `Integer`, the behavior is unspecified.
- 11 b) Let *L* be the match result attribute of the receiver, and let *i* be the value of *index*.
- 12 c) If *i* is smaller than 0, or larger than or equal to the number of elements of *L*, raise a  
13 direct instance of the class `IndexError`.
- 14 d) Let *S* and *b* be the substring and the index of the substring of the *i*th element of *L*,  
15 respectively. Let *e* be the sum of *b* and the length of *S*.
- 16 e) Return a new instance of the class `Array` which contains two instances of the class  
17 `Integer`, the one whose value is *b* and the other whose value is *e*, in this order.

### 18 15.2.16.3.8 MatchData#post\_match

---

19 `post_match`

---

20 **Visibility:** public

21 **Behavior:** The method returns an instance of the class `String` the content of which is the  
22 post-match of the receiver.

### 23 15.2.16.3.9 MatchData#pre\_match

---

24 `pre_match`

---

25 **Visibility:** public

26 **Behavior:** The method returns an instance of the class `String` the content of which is the  
27 pre-match of the receiver.

1 **15.2.16.3.10 MatchData#size**

---

2 **size**

---

3 **Visibility:** public

4 **Behavior:** Same as the method `length` (see 15.2.16.3.6).

5 **15.2.16.3.11 MatchData#string**

---

6 **string**

---

7 **Visibility:** public

8 **Behavior:**

9 The method returns an instance of the class `String` the content of which is the string  
10 attribute of the receiver.

11 **15.2.16.3.12 MatchData#to\_a**

---

12 **to\_a**

---

13 **Visibility:** public

14 **Behavior:**

- 15 a) Let  $L$  be the match result attribute of the receiver.
- 16 b) Create an empty direct instance  $A$  of the class `Array`.
- 17 c) For each element  $e$  of  $L$ , in the same order in the list, append to  $A$  an instance of the  
18 class `String` whose content is the substring of  $e$ .
- 19 d) Return  $A$ .

20 **15.2.16.3.13 MatchData#to\_s**

---

21 **to\_s**

---

22 **Visibility:** public

23 **Behavior:** The method returns an instance of the class `String` the content of which is the  
24 matched substring of the receiver.

1 **15.2.17 Proc**

2 **15.2.17.1 General description**

3 Instances of the class `Proc` represent *blocks*.

4 An instance of the class `Proc` has the following attribute.

5     **block:** The block represented by the instance.

6 **15.2.17.2 Direct superclass**

7 The class `Object`

8 **15.2.17.3 Singleton methods**

9 **15.2.17.3.1 Proc.new**

---

10 `Proc.new(&block)`

---

11 **Visibility:** public

12 **Behavior:**

13 a) If *block* is given, let *B* be *block*.

14 b) Otherwise:

15     1) If the top of `[[block]]` is block-not-given, then raise a direct instance of the class  
16         `ArgumentError`.

17     2) Otherwise, let *B* be the top of `[[block]]`.

18     c) Create a new direct instance of the class `Proc` which has *B* as its block attribute.

19     d) Return the instance.

20 **15.2.17.4 Instance methods**

21 **15.2.17.4.1 Proc#[]**

---

22 `[](*args)`

---

23 **Visibility:** public

24 **Behavior:** Same as the method `call` (see 15.2.17.4.3).

25 **15.2.17.4.2 Proc#arity**

---

1     arity

---

2     **Visibility:** public

3     **Behavior:** Let  $B$  be the block represented by the receiver.

4     a) If a *block-parameter* is omitted in  $B$ , return an instance of the class `Integer` whose  
5       value is implementation-defined.

6     b) If a *block-parameter* is present in  $B$ :

7       1) If a *block-parameter-list* is omitted in the *block-parameter*, return an instance of  
8          the class `Integer` whose value is 0.

9       2) If a *block-parameter-list* is present in the *block-parameter*:

10        i) If the *block-parameter-list* is of the form *left-hand-side*, return an instance of  
11          the class `Integer` whose value is 1.

12        ii) If the *block-parameter-list* is of the form *multiple-left-hand-side*:

13          I) If the *multiple-left-hand-side* is of the form *grouped-left-hand-side*, return  
14             an instance of the class `Integer` whose value is implementation-defined.

15          II) If the *multiple-left-hand-side* is of the form *packing-left-hand-side*, return  
16             an instance of the class `Integer` whose value is  $-1$ .

17          III) Otherwise, let  $n$  be the number of *multiple-left-hand-side-items* of the  
18             *multiple-left-hand-side*.

19          IV) If the *multiple-left-hand-side* ends with a *packing-left-hand-side*, return  
20             an instance of the class `Integer` whose value is  $-(n+1)$ .

21          V) Otherwise, return an instance of the class `Integer` whose value is  $n$ .

22     **15.2.17.4.3 Proc#call**

---

23     `call(*args)`

---

24     **Visibility:** public

25     **Behavior:** Let  $B$  be the block attribute of the receiver. Let  $L$  be an empty list.

26     a) Append each element of  $args$ , in the indexing order, to  $L$ .

27     b) Call  $B$  with  $L$  as the arguments (see 11.3.3). Let  $V$  be the result of the call.

28     c) Return  $V$ .

1 **15.2.17.4.4 Proc#clone**

---

2 **clone**

---

3 **Visibility:** public

4 **Behavior:**

- 5 a) Create a direct instance of the class of the receiver which has no bindings of instance  
6 variables. Let  $O$  be the newly created instance.
- 7 b) For each binding  $B$  of the instance variables of the receiver, create a variable binding  
8 with the same name and value as  $B$  in the set of bindings of instance variables of  $O$ .
- 9 c) If the receiver is associated with a singleton class, let  $E_o$  be the singleton class, and  
10 take the following steps:
- 11 1) Create a singleton class whose direct superclass is the direct superclass of  $E_o$ . Let  
12  $E_n$  be the singleton class.
- 13 2) For each binding  $B_{v1}$  of the constants of  $E_o$ , create a variable binding with the  
14 same name and value as  $B_{v1}$  in the set of bindings of constants of  $E_n$ .
- 15 3) For each binding  $B_{v2}$  of the class variables of  $E_o$ , create a variable binding with  
16 the same name and value as  $B_{v2}$  in the set of bindings of class variables of  $E_n$ .
- 17 4) For each binding  $B_m$  of the instance methods of  $E_o$ , create a method binding with  
18 the same name and value as  $B_m$  in the set of bindings of instance methods of  $E_n$ .
- 19 5) Associate  $O$  with  $E_n$ .
- 20 d) Set the block attribute of  $O$  to the block attribute of the receiver.
- 21 e) Return  $O$ .

22 **15.2.17.4.5 Proc#dup**

---

23 **dup**

---

24 **Visibility:** public

25 **Behavior:**

- 26 a) Create a direct instance of the class of the receiver which has no bindings of instance  
27 variables. Let  $O$  be the newly created instance.
- 28 b) Set the block attribute of  $O$  to the block attribute of the receiver.
- 29 c) Return  $O$ .

## 1 15.2.18 Struct

### 2 15.2.18.1 General description

3 The class `Struct` is a generator of a structure type which is a class defining a set of fields  
4 and methods for accessing these fields. Fields are indexed by integers starting from 0 (see  
5 15.2.18.3.1). An instance of a generated class has values for the set of fields. Those values can  
6 be referred to and updated with accessor methods for their fields.

### 7 15.2.18.2 Direct superclass

8 The class `Object`

### 9 15.2.18.3 Singleton methods

#### 10 15.2.18.3.1 `Struct.new`

---

11 `Struct.new( string, *symbol_list)`

---

12 **Visibility:** public

13 **Behavior:** The method creates a class defining a set of fields and accessor methods for  
14 these fields.

15 When the method is invoked, take the following steps:

- 16 a) Create a direct instance of the class `Class` which has the class `Struct` as its direct  
17 superclass. Let  $C$  be that class.
- 18 b) If  $string$  is not an instance of the class `String` or the class `Symbol`, the behavior is  
19 unspecified.
- 20 c) If  $string$  is an instance of the class `String`, let  $N$  be the content of the instance.
  - 21 1) If  $N$  is not of the form *constant-identifier*, raise a direct instance of the class  
22 `ArgumentError`.
  - 23 2) Otherwise,
    - 24 i) If the binding with name  $N$  exists in the set of bindings of constants in the  
25 class `Struct`, replace the value of the binding with  $C$ .
    - 26 ii) Otherwise, create a constant binding in the class `Struct` with name  $N$  and  
27 value  $C$ .
- 28 d) If  $string$  is an instance of the class `Symbol`, prepend the instance to  $symbol\_list$ .
- 29 e) Let  $i$  be 0.
- 30 f) For each element  $S$  of  $symbol\_list$ , take the following steps:

- 1) Let  $N$  be the name designated by  $S$ .
  - 2) Define a field, which is named  $N$  and is indexed by  $i$ , in  $C$ .
  - 3) If  $N$  is of the form *local-variable-identifier* or *constant-identifier*:
    - i) Define a method named  $N$  in  $C$  which takes no arguments, and when invoked, returns the value of the field named  $N$ .
    - ii) Define a method named  $N=$  (i.e.  $N$  postfixed by “=”) in  $C$  which takes one argument, and when invoked, sets the field named  $N$  to the given argument and returns the argument.
  - 4) Increment  $i$  by 1.
- g) Return  $C$ .

Classes created by the method `Struct.new` are equipped with public singleton methods `new`, `[]`, and `members`. The following describes those methods, assuming that the name of a class created by the method `Struct.new` is  $C$ .

---

`C.new(*args)`

---

**Visibility:** public

**Behavior:**

- a) Create a direct instance of the class with the set of fields the receiver defines. Let  $I$  be the instance.
- b) Invoke the method `initialize` on  $I$  with  $args$  as the list of arguments.
- c) Return  $I$ .

---

`C.[]( *args )`

---

**Visibility:** public

**Behavior:** Same as the method `new` described above.

---

`C.members`

---

**Visibility:** public

**Behavior:**

- 1 a) Create a direct instance  $A$  of the class `Array`. For each field of the receiver, in the  
2 indexing order of the fields, create a direct instance of the class `String` whose content  
3 is the name of the field and append the instance to  $A$ .
- 4 b) Return  $A$ .

#### 5 15.2.18.4 Instance methods

##### 6 15.2.18.4.1 `Struct#==`

---

7 `==(other)`

---

8 **Visibility:** public

9 **Behavior:**

- 10 a) If  $other$  and the receiver are the same object, return **true**.
- 11 b) If the class of  $other$  and that of the receiver are different, return **false**.
- 12 c) Otherwise, for each field named  $f$  of the receiver, take the following steps:
- 13 1) Let  $R$  and  $O$  be the values of the fields named  $f$  of the receiver and  $other$  respec-  
14 tively.
- 15 2) If  $R$  and  $O$  are not the same object,
- 16 i) Invoke the method `==` on  $R$  with  $O$  as the only argument. Let  $V$  be the  
17 resulting value of the invocation.
- 18 ii) If  $V$  is a falseish object, return **false**.
- 19 d) Return **true**.

##### 20 15.2.18.4.2 `Struct#[[]]`

---

21 `[](name)`

---

22 **Visibility:** public

23 **Behavior:**

- 24 a) If  $name$  is an instance of the class `Symbol` or the class `String`:
- 25 1) Let  $N$  be the name designated by  $name$ .
- 26 2) If the receiver has the field named  $N$ , return the value of the field.

- 1           3) Otherwise, let  $S$  be an instance of the class `Symbol` with name  $N$  and raise a direct  
2 instance of the class `NameError` which has  $S$  as its name attribute.
- 3           b) If  $name$  is an instance of the class `Integer`, let  $i$  be the value of  $name$ . Let  $n$  be the  
4 number of the fields of the receiver.
  - 5           1) If  $i$  is negative, let new  $i$  be  $n + i$ .
  - 6           2) If  $i$  is still negative or  $i$  is larger than or equal to  $n$ , raise a direct instance of the  
7 class `IndexError`.
  - 8           3) Otherwise, return the value of the field whose index is  $i$ .
- 9           c) Otherwise, the behavior of the method is unspecified.

#### 10 **15.2.18.4.3 Struct#[]=**

---

11       []=(*name*, *obj*)

---

12       **Visibility:** public

13       **Behavior:**

- 14       a) If  $name$  is an instance of the class `Symbol` or an instance of the class `String`:
  - 15           1) Let  $N$  be the name designated by  $name$ .
  - 16           2) If the receiver has the field named  $N$ ,
    - 17            i) Replace the value of the field with  $obj$ ,
    - 18            ii) Return  $obj$ .
  - 19           3) Otherwise, let  $S$  be an instance of the class `Symbol` with name  $N$  and raise a direct  
20 instance of the class `NameError` which has  $S$  as its name attribute.
- 21       b) If  $name$  is an instance of the class `Integer`, let  $i$  be the value of  $name$ . Let  $n$  be the  
22 number of the fields of the receiver.
  - 23           1) If  $i$  is negative, let new  $i$  be  $n + i$ .
  - 24           2) If  $i$  is still negative or  $i$  is larger than or equal to  $n$ , raise a direct instance of the  
25 class `IndexError`.
  - 26           3) Otherwise,
    - 27            i) Replace the value of the field whose index is  $i$  with  $obj$
    - 28            ii) Return  $obj$ .
- 29       c) Otherwise, the behavior of the method is unspecified.

#### 1 15.2.18.4.4 Struct#each

---

2 each(&block)

---

3 **Visibility:** public

4 **Behavior:**

- 5 a) If *block* is not given, the behavior is unspecified.
- 6 b) For each field of the receiver, in the indexing order, call *block* with the value of the  
7 field as the only argument.
- 8 c) Return the receiver.

#### 9 15.2.18.4.5 Struct#each\_pair

---

10 each\_pair(&block)

---

11 **Visibility:** public

12 **Behavior:**

- 13 a) If *block* is not given, the behavior is unspecified.
- 14 b) For each field of the receiver, in the indexing order, take the following steps:
- 15 1) Let *N* and *V* be the name and the value of the field respectively. Let *S* be an  
16 instance of the class `Symbol` with name *N*.
- 17 2) Call *block* with the list of arguments which contains *S* and *V* in this order.
- 18 c) Return the receiver.

#### 19 15.2.18.4.6 Struct#members

---

20 members

---

21 **Visibility:** public

22 **Behavior:** Same as the method `members` described in 15.2.18.3.1.

#### 23 15.2.18.4.7 Struct#select

---

1     `select(&block)`

---

2     **Visibility:** public

3     **Behavior:**

- 4     a) If *block* is not given, the behavior is unspecified.
- 5     b) Create an empty direct instance of the class `Array`. Let *A* be the instance.
- 6     c) For each field of the receiver, in the indexing order, take the following steps:
- 7         1) Let *V* be the value of the field.
- 8         2) Call *block* with *V* as the only argument. Let *R* be the resulting value of the call.
- 9         3) If *R* is a trueish object, append *V* to *A*.
- 10    d) Return *A*.

11 **15.2.18.4.8 Struct#initialize**

---

12     `initialize(*args)`

---

13     **Visibility:** private

14     **Behavior:** Let  $N_a$  be the length of *args*, and let  $N_f$  be the number of the fields of the  
15     receiver.

- 16     a) If  $N_a$  is larger than  $N_f$ , raise a direct instance of the class `ArgumentError`.
- 17     b) Otherwise, for each field *f* of the receiver, let *i* be the index of *f*, and set the value of *f*  
18     to the *i*th element of *args*, or to `nil` when *i* is equal to or larger than  $N_a$ .
- 19     c) Return an implementation-defined value.

20 **15.2.18.4.9 Struct#initialize\_copy**

---

21     `initialize_copy(original)`

---

22     **Visibility:** private

23     **Behavior:**

- 24     a) If the receiver and *original* are the same object, return an implementation-defined  
25     value.

- 1        b) If *original* is not an instance of the class of the receiver, raise a direct instance of the  
2            class `TypeError`.
- 3        c) If the number of the fields of the receiver and the number of the fields of *original* are  
4            different, raise a direct instance of the class `TypeError`.
- 5        d) For each field *f* of *original*, let *i* be the index of *f*, and set the value of the *i*th field of  
6            the receiver to the value of *f*.
- 7        e) Return an implementation-defined value.

## 8    **15.2.19    Time**

### 9    **15.2.19.1    General description**

10 Instances of the class `Time` represent dates and times.

11 An instance of the class `Time` holds the following attributes.

12        **Microseconds:** Microseconds since January 1, 1970 00:00 UTC. Microseconds is an integer  
13            whose range is implementation-defined. The value of microseconds attributes is rounded  
14            to fit in the representation of microseconds in an implementation-defined way. If an out of  
15            range value is given as microseconds when creating an instance of the class `Time`, a direct  
16            instance of either of the class `ArgumentError` or the class `RangeError` shall be raised.  
17            Which class is chosen is implementation-defined.

18        **Time zone:** The time zone.

### 19    **15.2.19.2    Direct superclass**

20 The class `Object`

### 21    **15.2.19.3    Time computation**

22 Mathematical functions introduced in this subclass are used throughout the descriptions in  
23 15.2.19. These functions are assumed to compute exact mathematical results using mathematical  
24 real numbers.

25 Leap seconds are ignored in this document. However, a conforming processor may support leap  
26 seconds in an implementation-defined way.

### 27    **15.2.19.3.1    Day**

28 The number of microseconds of a day is computed as follows:

$$MicroSecPerDay = 24 \times 60 \times 60 \times 10^6$$

29 The number of days since January 1, 1970 00:00 UTC which corresponds to microseconds *t* is  
30 computed as follows:

$$Day(t) = floor\left(\frac{t}{MicroSecPerDay}\right)$$

$floor(t)$  = The integer  $x$  such that  $x \leq t < x + 1$

1 The weekday which corresponds to microseconds  $t$  is computed as follows:

$$WeekDay(t) = (Day(t) + 4) \text{ modulo } 7$$

## 2 15.2.19.3.2 Year

3 A year has 365 days, except for leap years, which have 366 days. Leap years are those which  
4 are either:

- 5 • divisible by 4 and not divisible by 100, or
- 6 • divisible by 400.

7 The number of days from January 1, 1970 00:00 UTC to the beginning of a year  $y$  is computed  
8 as follows:

$$DayFromYear(y) = 365 \times (y - 1970) + floor\left(\frac{y - 1969}{4}\right) - floor\left(\frac{y - 1901}{100}\right) + floor\left(\frac{y - 1601}{400}\right)$$

9 Microseconds elapsed since January 1, 1970 00:00 UTC until the beginning of  $y$  is computed as  
10 follows:

$$MicroSecFromYear(y) = DayFromYear(y) \times MicroSecPerDay$$

11 The year number  $y$  which corresponds to microseconds  $t$  measured from January 1, 1970 00:00  
12 UTC is computed as follows.

$$YearFromTime(t) = y \text{ such that, } MicroSecFromYear(y - 1) < t \leq MicroSecFromYear(y)$$

13 The number of days from the beginning of the year for the given microseconds  $t$  is computed as  
14 follows.

$$DayWithinYear(t) = Day(t) - DayFromYear(YearFromTime(t))$$

1 **15.2.19.3.3 Month**

2 Months have usual number of days. Leap years have the extra day in February. Each month is  
 3 identified by the number in the range 1 to 12, in the order from January to December.

4 The month number which corresponds to microseconds  $t$  measured from January 1, 1970 00:00  
 5 UTC is computed as follows.

$$MonthFromTime(t) = \begin{cases} 1 & \text{if } 0 \leq DayWithinYear(t) < 31 \\ 2 & \text{if } 31 \leq DayWithinYear(t) < 59 + LeapYear(t) \\ 3 & \text{if } 59 + LeapYear(t) \leq DayWithinYear(t) < 90 + LeapYear(t) \\ 4 & \text{if } 90 + LeapYear(t) \leq DayWithinYear(t) < 120 + LeapYear(t) \\ 5 & \text{if } 120 + LeapYear(t) \leq DayWithinYear(t) < 151 + LeapYear(t) \\ 6 & \text{if } 151 + LeapYear(t) \leq DayWithinYear(t) < 181 + LeapYear(t) \\ 7 & \text{if } 181 + LeapYear(t) \leq DayWithinYear(t) < 212 + LeapYear(t) \\ 8 & \text{if } 212 + LeapYear(t) \leq DayWithinYear(t) < 243 + LeapYear(t) \\ 9 & \text{if } 243 + LeapYear(t) \leq DayWithinYear(t) < 273 + LeapYear(t) \\ 10 & \text{if } 273 + LeapYear(t) \leq DayWithinYear(t) < 304 + LeapYear(t) \\ 11 & \text{if } 304 + LeapYear(t) \leq DayWithinYear(t) < 334 + LeapYear(t) \\ 12 & \text{if } 334 + LeapYear(t) \leq DayWithinYear(t) < 365 + LeapYear(t) \end{cases}$$

$$LeapYear(t) = \begin{cases} 1 & \text{if } YearFromTime(t) \text{ is a leap year} \\ 0 & \text{otherwise} \end{cases}$$

6 **15.2.19.3.4 Days of month**

7 The day of the month which corresponds to microseconds  $t$  measured from January 1, 1970  
 8 00:00 UTC is computed as follows.

$$DayWithinMonth(t) = \begin{cases} DayWithinYear(t) + 1 & \text{if } MonthFromTime(t) = 1 \\ DayWithinYear(t) - 30 & \text{if } MonthFromTime(t) = 2 \\ DayWithinYear(t) - 58 - LeapYear(t) & \text{if } MonthFromTime(t) = 3 \\ DayWithinYear(t) - 89 - LeapYear(t) & \text{if } MonthFromTime(t) = 4 \\ DayWithinYear(t) - 119 - LeapYear(t) & \text{if } MonthFromTime(t) = 5 \\ DayWithinYear(t) - 150 - LeapYear(t) & \text{if } MonthFromTime(t) = 6 \\ DayWithinYear(t) - 180 - LeapYear(t) & \text{if } MonthFromTime(t) = 7 \\ DayWithinYear(t) - 211 - LeapYear(t) & \text{if } MonthFromTime(t) = 8 \\ DayWithinYear(t) - 242 - LeapYear(t) & \text{if } MonthFromTime(t) = 9 \\ DayWithinYear(t) - 272 - LeapYear(t) & \text{if } MonthFromTime(t) = 10 \\ DayWithinYear(t) - 303 - LeapYear(t) & \text{if } MonthFromTime(t) = 11 \\ DayWithinYear(t) - 333 - LeapYear(t) & \text{if } MonthFromTime(t) = 12 \end{cases}$$

1 **15.2.19.3.5 Hours, Minutes, and Seconds**

2 The numbers of microseconds in an hour, a minute, and a second are as follows:

$$\begin{aligned} \text{MicroSecPerHour} &= 60 \times 60 \times 10^6 \\ \text{MicroSecPerMinute} &= 60 \times 10^6 \\ \text{MicroSecPerSecond} &= 10^6 \end{aligned}$$

3 The hour, the minute, and the second which correspond to microseconds  $t$  measured from  
4 January 1, 1970 00:00 UTC are computed as follows.

$$\begin{aligned} \text{HourFromTime}(t) &= \text{floor} \left( \frac{t}{\text{MicroSecPerHour}} \right) \text{ modulo } 24 \\ \text{MinuteFromTime}(t) &= \text{floor} \left( \frac{t}{\text{MicroSecPerMinute}} \right) \text{ modulo } 60 \\ \text{SecondFromTime}(t) &= \text{floor} \left( \frac{t}{\text{MicroSecPerSecond}} \right) \text{ modulo } 60 \end{aligned}$$

5 **15.2.19.4 Time zone and Local time**

6 The current time zone is determined from time zone information provided by the underlying  
7 system. If the system does not provide information on the current local time zone, the time  
8 zone attribute of an instance of the class `Time` is implementation-defined.

9 The local time for an instance of the class `Time` is computed from its microseconds  $t$  and time  
10 zone  $z$  as follows.

$$\begin{aligned} \text{LocalTime} &= t + \text{ZoneOffset}(z) \\ \text{ZoneOffset}(z) &= \text{UTC offset of } z \text{ measured in microseconds} \end{aligned}$$

11 **15.2.19.5 Daylight saving time**

12 On system where it is possible to determine the daylight saving time for each time zone, a  
13 conforming processor should adjust the microseconds attributes of an instance of the class `Time`  
14 if that microseconds falls within the daylight saving time of the time zone attributes of the  
15 instance. An algorithm used for the adjustment is implementation-defined.

16 **15.2.19.6 Singleton methods**

17 **15.2.19.6.1 Time.at**

---

1 `Time.at(*args)`

---

2 **Visibility:** public

3 **Behavior:**

4 a) If the length of *args* is 0 or larger than 2, raise a direct instance of the class `ArgumentError`.

5 b) If the length of *args* is 1, let *A* be the only argument.

6 1) If *A* is an instance of the class `Time`, return a new instance of the class `Time` which  
7 represents the same time and has the same time zone as *A*.

8 2) If *A* is an instance of the class `Integer` or an instance of the class `Float`:

9 i) Let *N* be the value of *A*.

10 ii) Create a direct instance of the class `Time` which represents the time at  $N \times 10^6$   
11 microseconds since January 1, 1970 00:00 UTC, with the current local time  
12 zone.

13 iii) Return the resulting instance.

14 3) Otherwise, the behavior is unspecified.

15 c) If the length of *args* is 2, let *S* and *M* be the first and second element of *args*.

16 1) If *S* is an instance of the class `Integer`, let  $N_S$  be the value of *S*.

17 2) Otherwise, the behavior is unspecified.

18 3) If *M* is an instance of the class `Integer` or an instance of the class `Float`, let  $N_M$   
19 be the value of *M*.

20 4) Otherwise, the behavior is unspecified.

21 5) Create a direct instance of the class `Time` which represents the time at  $N_S \times 10^6 +$   
22  $N_M$  microseconds since January 1, 1970 00:00 UTC, with the current local time  
23 zone.

24 6) Return the resulting instance.

25 **15.2.19.6.2 Time.gm**

---

26 `Time.gm(year, month=1, day=1, hour=0, min=0, sec=0, usec=0)`

---

27 **Visibility:** public

28 **Behavior:**

1 a) Compute an integer value for *year*, *day*, *hour*, *min*, *sec*, and *usec* as described below.  
2 Let *Y*, *D*, *H*, *Min*, *S*, and *U* be integers thus converted.

3 An integer *I* is determined from the given object *O* as follows:

- 4 1) If *O* is an instance of the class **Integer**, let *I* be the value of *O*.
- 5 2) If *O* is an instance of the class **Float**, let *I* be the integral part of the value of *O*.
- 6 3) If *O* is an instance of the class **String**:
  - 7 i) If the content of *O* is a sequence of *decimal-digits*, let *I* be the value of those  
8 sequence of digits computed using base 10.
  - 9 ii) Otherwise, the behavior is unspecified.
- 10 4) Otherwise, the behavior is unspecified.

11 b) Compute an integer value from *month* as follows:

- 12 1) If *month* is an instance of the class **Integer**, let *Mon* be the value of *month*.
- 13 2) If *month* is an instance of the class **String**:
  - 14 i) If the content of *month* is the same as one of the names of the months in the  
15 lower row on Table 5, ignoring the differences in case, let *Mon* be the integer  
16 which corresponds to *month* in the upper row on the same table.
  - 17 ii) If the first character of *month* is *decimal-digit*, compute an integer value from  
18 *month* as in Step a). Let *Mon* be the resulting integer.
  - 19 iii) Otherwise, raise a direct instance of the class **ArgumentError**.
- 20 3) Otherwise, the behavior is unspecified.

21 c) If *Y* is an integer such that  $0 \leq Y \leq 138$ , the behavior is implementation-defined.

22 d) If each integer computed above is outside the range as listed below, raise a direct  
23 instance of the class **ArgumentError**.

- 24 •  $1 \leq Mon \leq 12$
- 25 •  $1 \leq D \leq 31$
- 26 •  $0 \leq H \leq 23$
- 27 •  $0 \leq Min \leq 59$
- 28 •  $0 \leq S \leq 60$

29 Whether any conditions are placed on *Y* is implementation-defined.

- 1 e) Let  $t$  be an integer which satisfies all of the following equations.
- 2     •  $YearFromTime(t) = Y$
- 3     •  $MonthFromTime(t) = Mon$
- 4     •  $DayWithinMonth(t) = 1$
- 5 f) Compute microseconds  $T$  as follows.

$$T = t + D \times MicroSecPerDay + H \times MicroSecPerHour + \\ Min \times MicroSecPerMinute + S \times 10^6 + U$$

- 6 g) Create a direct instance of the class `Time` which represents the time at  $T$  since January
- 7 1, 1970 00:00 UTC, with the UTC time zone.
- 8 h) Return the resulting instance.

**Table 5 – The names of months and corresponding integer**

1	2	3	4	5	6	7	8	9	10	11	12
Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec

9 **15.2.19.6.3 Time.local**

---

```
Time.local( year, month=1, day=1, hour=0, min=0, sec=0, usec=0 )
```

---

11 **Visibility:** public

12 **Behavior:** Same as the method `Time.gm` (see 15.2.19.6.2), except that the method returns

13 a direct instance of the class `Time` which has the current local time zone as its time zone.

14 **15.2.19.6.4 Time.mktime**

---

```
Time.mktime( year, month=1, day=1, hour=0, min=0, sec=0, usec=0 )
```

---

16 **Visibility:** public

17 **Behavior:** Same as the method `Time.local` (see 15.2.19.6.3).

18 **15.2.19.6.5 Time.now**

---

1 `Time.now`

---

2 **Visibility:** public

3 **Behavior:** This method returns a direct instance of the class `Time` which represents the  
4 current time with the current local time zone.

5 The behavior of this method is the same as the method `new` (see 15.2.3.3.3).

6 **15.2.19.6.6 Time.utc**

---

7 `Time.utc( year, month=1, day=1, hour=0, min=0, sec=0, usec=0 )`

---

8 **Visibility:** public

9 **Behavior:** Same as the method `Time.gm` (see 15.2.19.6.2).

10 **15.2.19.7 Instance methods**

11 **15.2.19.7.1 Time#+**

---

12 `+( offset )`

---

13 **Visibility:** public

14 **Behavior:**

- 15 a) If *offset* is not an instance of the class `Integer` or the class `Float`, the behavior is  
16 unspecified.
- 17 b) Let *V* be the value of *offset*.
- 18 c) Let *o* be the result of computing  $V \times 10^6$ .
- 19 d) Let *t* and *z* be the microseconds and time zone attribute of the receiver.
- 20 e) Create a direct instance of the class `Time` which represents the time at (*t* + *o*) microsec-  
21 onds since January 1, 1970 00:00 UTC, with *z* as its time zone.
- 22 f) Return the resulting instance.

23 **15.2.19.7.2 Time#-**

---

24 `-( offset )`

---

25 **Visibility:** public

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28

**Behavior:**

- a) If *offset* is not an instance of the class `Integer` or the class `Float`, the behavior is unspecified.
- b) Let *V* be the value of *offset*.
- c) Let *o* be the result of computing  $V \times 10^6$ .
- d) Let *t* and *z* be the microseconds and time zone attribute of the receiver.
- e) Create a direct instance of the class `Time` which represents the time at  $t - o$  microseconds since January 1, 1970 00:00 UTC, with *z* as its time zone.
- f) Return the resulting instance.

**15.2.19.7.3 Time#<=>**

---

<=>( *other* )

---

**Visibility:** public

**Behavior:**

- a) If *other* is not an instance of the class `Time`, return `nil`.
- b) Otherwise, let  $T_r$  and  $T_o$  be microseconds attributes of the receiver and *other*, respectively.
  - 1) If  $T_r > T_o$ , return an instance of the class `Integer` whose value is 1.
  - 2) If  $T_r = T_o$ , return an instance of the class `Integer` whose value is 0.
  - 3) If  $T_r < T_o$ , return an instance of the class `Integer` whose value is -1.

**15.2.19.7.4 Time#asctime**

---

asctime

---

**Visibility:** public

**Behavior:**

- a) Compute the local time from the receiver (see 15.2.19.4). Let *t* be the result.
- b) Let *W* be the name of the day of the week in the second row on Table 6 which corresponds to *WeekDay(t)* in the upper row on the same table.
- c) Let *Mon* be the name of the month in the second row on Table 5 which corresponds to *MonthFromTime(t)* in the upper row on the same table.

1 d) Let  $D$ ,  $H$ ,  $M$ ,  $S$ , and  $Y$  be as follows:

$D = \text{DayWithinMonth}(t)$

$H = \text{HourFromTime}(t)$

$M = \text{MinuteFromTime}(t)$

$S = \text{SecondFromTime}(t)$

$Y = \text{YearFromTime}(t)$

2 e) Create a direct instance of the class `String`, the content of which is the following  
3 sequence of characters:

$W \text{ Mon } D \text{ H:M:S } Y$

4  $D$  is formatted as two digits with a leading space character (0x20) as necessary.  $H$ ,  $M$ ,  
5 and  $S$  are formatted as two digits with a leading zero as necessary.

6 EXAMPLE `Time.local(2001, 10, 1, 13, 20, 5).asctime` returns "Mon Oct 1 13:20:05 2001".

7 f) Return the resulting instance.

**Table 6 – The names of the days of the week corresponding to integers**

0	1	2	3	4	5	6
Sun	Mon	Tue	Wed	Thu	Fry	Sat

8 **15.2.19.7.5 Time#ctime**

---

9 `ctime`

---

10 **Visibility:** public

11 **Behavior:** Same as the method `asctime` (see 15.2.19.7.4).

12 **15.2.19.7.6 Time#day**

---

13 `day`

---

14 **Visibility:** public

15 **Behavior:**

16 a) Compute the local time from the receiver (see 15.2.19.4). Let  $t$  be the result.

17 b) Compute  $\text{DayWithinMonth}(t)$ .

18 c) Return an instance of the class `Integer` whose value is the result of Step b).

1 **15.2.19.7.7 Time#dst?**

---

2 dst?

---

3 **Visibility:** public

4 **Behavior:** Let  $T$  and  $Z$  be the microseconds and time zone attribute of the receiver.

5 a) If  $T$  falls within the daylight saving time of  $Z$ , return **true**.

6 b) Otherwise, return **false**.

7 **15.2.19.7.8 Time#getgm**

---

8 getgm

---

9 **Visibility:** public

10 **Behavior:** Same as the method `getutc` (see 15.2.19.7.10).

11 **15.2.19.7.9 Time#getlocal**

---

12 getlocal

---

13 **Visibility:** public

14 **Behavior:** The method returns a new instance of the class `Time` which has the same  
15 microseconds as the receiver, but has current local time zone as its time zone.

16 **15.2.19.7.10 Time#getutc**

---

17 getutc

---

18 **Visibility:** public

19 **Behavior:** The method returns a new instance of the class `Time` which has the same  
20 microseconds as the receiver, but has UTC as its time zone.

21 **15.2.19.7.11 Time#gmt?**

---

22 gmt?

---

23 **Visibility:** public

24 **Behavior:** Same as the method `utc?` (see 15.2.19.7.28).

1 **15.2.19.7.12 Time#gmt\_offset**

---

2 `gmt_offset`

---

3 **Visibility:** public

4 **Behavior:** Same as the method `utc_offset` (see 15.2.19.7.29).

5 **15.2.19.7.13 Time#gmtime**

---

6 `gmtime`

---

7 **Visibility:** public

8 **Behavior:** Same as the method `utc` (see 15.2.19.7.27).

9 **15.2.19.7.14 Time#gmtoff**

---

10 `gmtoff`

---

11 **Visibility:** public

12 **Behavior:** Same as the method `utc_offset` (see 15.2.19.7.29).

13 **15.2.19.7.15 Time#hour**

---

14 `hour`

---

15 **Visibility:** public

16 **Behavior:**

17 a) Compute the local time from the receiver (see 15.2.19.4). Let  $t$  be the result.

18 b) Compute  $HourFromTime(t)$ .

19 c) Return an instance of the class `Integer` whose value is the result of Step b).

20 **15.2.19.7.16 Time#initialize**

---

21 `initialize`

---

22 **Visibility:** private

1     **Behavior:**

- 2     a) Set the microseconds attribute of the receiver to microseconds elapsed since January  
3       1, 1970 00:00 UTC.
- 4     b) Set the time zone attribute of the receiver to the current local time zone.
- 5     c) Return an implementation-defined value.

6     **15.2.19.7.17 Time#initialize\_copy**

---

7     initialize\_copy(*original*)

---

8     **Visibility:** private

9     **Behavior:**

- 10    a) If *original* is not an instance of the class **Time**, raise a direct instance of the class  
11       **TypeError**.
- 12    b) Set the microseconds attribute of the receiver to the microseconds attribute of *original*.
- 13    c) Set the time zone attribute of the receiver to the time zone attribute of *original*.
- 14    d) Return an implementation-defined value.

15    **15.2.19.7.18 Time#localtime**

---

16    localtime

---

17    **Visibility:** public

18    **Behavior:**

- 19    a) Change the time zone attribute of the receiver to the current local time zone.
- 20    b) Return the receiver.

21    **15.2.19.7.19 Time#mday**

---

22    mday

---

23    **Visibility:** public

24    **Behavior:**

- 25    a) Compute the local time from the receiver (see 15.2.19.4). Let *t* be the result.

- 1        b) Compute *DayWithinMonth(t)*.  
2        c) Return an instance of the class `Integer` whose value is the result of Step b).

3 **15.2.19.7.20 Time#min**

---

4        `min`

---

5        **Visibility:** public

6        **Behavior:**

- 7        a) Compute the local time from the receiver (see 15.2.19.4). Let  $t$  be the result.  
8        b) Compute *MinuteFromTime(t)*.  
9        c) Return an instance of the class `Integer` whose value is the result of Step b).

10 **15.2.19.7.21 Time#mon**

---

11        `mon`

---

12        **Visibility:** public

13        **Behavior:**

- 14        a) Compute the local time from the receiver (see 15.2.19.4). Let  $t$  be the result.  
15        b) Compute *MonthFromTime(t)*.  
16        c) Return an instance of the class `Integer` whose value is the result of Step b).

17 **15.2.19.7.22 Time#month**

---

18        `month`

---

19        **Visibility:** public

20        **Behavior:** Same as the method `mon` (see 15.2.19.7.21).

21 **15.2.19.7.23 Time#sec**

---

22        `sec`

---

23        **Visibility:** public

1 **Behavior:**

- 2 a) Compute the local time from the receiver (see 15.2.19.4). Let  $t$  be the result.  
3 b) Compute  $SecondFromTime(t)$ .  
4 c) Return an instance of the class `Integer` whose value is the result of Step b).

5 **15.2.19.7.24 Time#to\_f**

---

6 `to_f`

---

7 **Visibility:** public

8 **Behavior:** Let  $t$  the microseconds attribute of the receiver.

- 9 a) Compute  $t/10^6$ .  
10 b) Return a direct instance of the class `Float` whose value is the result of Step a).

11 **15.2.19.7.25 Time#to\_i**

---

12 `to_i`

---

13 **Visibility:** public

14 **Behavior:** Let  $t$  the microseconds attribute of the receiver.

- 15 a) Compute  $\text{floor}(t/10^6)$ .  
16 b) Return an instance of the class `Integer` whose value is the result of Step a).

17 **15.2.19.7.26 Time#usec**

---

18 `usec`

---

19 **Visibility:** public

20 **Behavior:**

- 21 a) Compute the local time from the receiver (see 15.2.19.4). Let  $t$  be the result.  
22 b) Compute  $t$  modulo  $10^6$ .  
23 c) Return an instance of the class `Integer` whose value is the result of Step b).

24 **15.2.19.7.27 Time#utc**

---

1     **utc**

---

2     **Visibility:** public

3     **Behavior:**

4     a) Change the time zone attribute of the receiver to UTC.

5     b) Return the receiver.

6     **15.2.19.7.28 Time#utc?**

---

7     **utc?**

---

8     **Visibility:** public

9     **Behavior:** Let  $Z$  be the time zone attribute of the receiver.

10    a) If  $Z$  is UTC, return **true**.

11    b) Otherwise, return **false**.

12    **15.2.19.7.29 Time#utc\_offset**

---

13    **utc\_offset**

---

14    **Visibility:** public

15    **Behavior:** Let  $Z$  be the time zone attribute of the receiver.

16    a) Compute  $\text{floor}(\text{ZoneOffset}(Z)/10^6)$ .

17    b) Return an instance of the class `Integer` whose value is the result of Step a).

18    **15.2.19.7.30 Time#wday**

---

19    **wday**

---

20    **Visibility:** public

21    **Behavior:**

22    a) Compute the local time from the receiver (see 15.2.19.4). Let  $t$  be the result.

23    b) Compute  $\text{WeekDay}(t)$ .

24    c) Return an instance of the class `Integer` whose value is the result of Step b)

1 **15.2.19.7.31 Time#yday**

---

2 yday

---

3 **Visibility:** public

4 **Behavior:**

- 5 a) Compute the local time from the receiver (see 15.2.19.4). Let  $t$  be the result.
- 6 b) Compute *DayWithinYear*( $t$ ).
- 7 c) Return an instance of the class `Integer` whose value is the result of Step b).

8 **15.2.19.7.32 Time#year**

---

9 year

---

10 **Visibility:** public

11 **Behavior:**

- 12 a) Compute the local time from the receiver (see 15.2.19.4). Let  $t$  be the result.
- 13 b) Compute *YearFromTime*( $t$ ).
- 14 c) Return an instance of the class `Integer` whose value is the result of Step b).

15 **15.2.19.7.33 Time#zone**

---

16 zone

---

17 **Visibility:** public

18 **Behavior:** Let  $Z$  be the time zone attribute of the receiver.

- 19 a) Create a direct instance of the class `String`, the content of which represents the name  
20 of  $Z$ . The exact content of the instance is implementation-defined.
- 21 b) Return the resulting instance.

22 **15.2.20 IO**

23 **15.2.20.1 General description**

24 An instance of the class `IO` represents a stream, which is a source and/or a sink of data.

25 An instance of the class `IO` has the following attributes:

1     **readability flag:** A boolean value which indicates whether the stream can handle input  
2     operations.

3     An instance of the class `IO` is said to be **readable** if and only if this flag is true.

4     Reading from a stream which is not readable raises a direct instance of the class `IOError`.

5     **writability flag:** A boolean value which indicates whether the stream can handle output  
6     operations.

7     An instance of the class `IO` is said to be **writable** if and only if this flag is true.

8     Writing to a stream which is not writable raises a direct instance of the class `IOError`.

9     **openness flag:** A boolean value which indicates whether the stream is open.

10     An instance of the class `IO` is said to be **open** if and only if this flag is true. An instance  
11     of the class `IO` is said to be **closed** if and only if this flag is false.

12     A closed stream is neither readable nor writable.

13     **buffering flag:** A boolean value which indicates whether the data to be written to the  
14     stream is buffered.

15     When this flag is true, the output to the receiver may be delayed until the instance methods  
16     `flush` or `close` is invoked.

17     An instance of the class `SystemCallError` may be raised when the underlying system reported  
18     an error during the execution of methods of the class `IO`.

19     The behavior of the method `initialize` of the class `IO` is unspecified, i.e. whether a direct  
20     instance of the class `IO` other than the constants `STDIN`, `STDOUT` and `STDERR` of the class  
21     `Object` (see 15.2.1) can be created is unspecified.

22     NOTE   Note that an instance of the class `File`, which is a subclass of the class `IO`, can be created by  
23     the method `new` because the behavior of the method `initialize` is specified in 15.2.21.4.1.

24     In the following description of the methods of the class `IO`, a **byte** means an integer from 0 to  
25     255.

## 26   15.2.20.2   Direct superclass

27   The class `Object`

## 28   15.2.20.3   Included modules

29   The following module is included in the class `IO`.

- 30   •   `Enumerable`

## 1 15.2.20.4 Singleton methods

### 2 15.2.20.4.1 IO.open

---

3 `IO.open(*args, &block)`

---

4 **Visibility:** public

5 **Behavior:**

- 6 a) Invoke the method `new` on the receiver with all the elements of `args` as the arguments.  
7 Let `I` be the resulting value.
- 8 b) If `block` is not given, return `I`.
- 9 c) Otherwise, call `block` with `I` as the argument. Let `V` be the resulting value.
- 10 d) Invoke the method `close` (see 15.2.20.5.1) on `I` with no arguments, even when an  
11 exception is raised and not handled in Step c).
- 12 e) Return `V`.

13 **EXAMPLE** If `block` is given, the method `close` is invoked automatically.

```
14     File.open("data.txt"){|f|  
15         puts f.read  
16     }  
17
```

18 If `block` is not given, the method `close` should be invoked explicitly.

```
19     f = File.open("data.txt")  
20     puts f.read  
21     f.close  
22
```

23 **NOTE** The behavior of invoking the method `new` on the class `IO` is unspecified. Therefore, the  
24 behavior of invoking the method `open` on the class `IO` is also unspecified; however, the method `open`  
25 can be invoked on the class `File`, which is a subclass of the class `IO`.

## 26 15.2.20.5 Instance methods

### 27 15.2.20.5.1 IO#close

---

28 `close`

---

29 **Visibility:** public

30 **Behavior:**

- 31 a) If the receiver is closed, raise a direct instance of the class `IOError`.

- 1 b) If the buffering flag of the receiver is true, and the receiver is buffering any output,
- 2 immediately write all the buffered data to the stream which the receiver represents.
- 3 c) Set the openness flag of the receiver to false.
- 4 d) Return an implementation-defined value.

#### 5 **15.2.20.5.2 IO#closed?**

---

6 `closed?`

---

7 **Visibility:** public

8 **Behavior:**

- 9 a) If the receiver is closed, return **true**.
- 10 b) Otherwise, return **false**.

#### 11 **15.2.20.5.3 IO#each**

---

12 `each(&block)`

---

13 **Visibility:** public

14 **Behavior:**

- 15 a) If *block* is not given, the behavior is unspecified.
- 16 b) If the receiver is not readable, raise a direct instance of the class `IOError`.
- 17 c) If the receiver has reached its end, return the receiver.
- 18 d) Otherwise, read characters from the receiver until `0x0a` is read or the receiver reaches
- 19 its end.
- 20 e) Create a direct instance of the class `String` whose content is the sequence of characters
- 21 read in Step d). Call *block* with this instance as an argument.
- 22 f) Continue processing from Step c).

#### 23 **15.2.20.5.4 IO#each\_byte**

---

24 `each_byte(&block)`

---

25 **Visibility:** public

- 1     **Behavior:**
- 2     a) If *block* is not given, the behavior is unspecified.
- 3     b) If the receiver is not readable, raise a direct instance of the class `IOError`.
- 4     c) If the receiver has reached its end, return the receiver.
- 5     d) Otherwise, read a single byte from the receiver. Call *block* with an argument, an  
6         instance of the class `Integer` whose value is the byte.
- 7     e) Continue processing from Step c).

8     **15.2.20.5.5 IO#each\_line**

---

9     `each_line(&block)`

---

10    **Visibility:** public

11    **Behavior:** Same as the method `each` (see 15.2.20.5.3).

12    **15.2.20.5.6 IO#eof?**

---

13    `eof?`

---

14    **Visibility:** public

15    **Behavior:**

- 16    a) If the receiver is not readable, raise a direct instance of the class `IOError`.
- 17    b) If the receiver has reached its end, return **true**. Otherwise, return **false**.

18    **15.2.20.5.7 IO#flush**

---

19    `flush`

---

20    **Visibility:** public

21    **Behavior:**

- 22    a) If the receiver is not writable, raise a direct instance of the class `IOError`.
- 23    b) If the buffering flag of the receiver is true, and the receiver is buffering any output,  
24         immediately write all the buffered data to the stream which the receiver represents.
- 25    c) Return the receiver.

1 **15.2.20.5.8 IO#getc**

---

2 `getc`

---

3 **Visibility:** public

4 **Behavior:**

- 5 a) If the receiver is not readable, raise a direct instance of the class `IOError`.
- 6 b) If the receiver has reached its end, return `nil`.
- 7 c) Otherwise, read a character from the receiver. Return an instance of the class `Object`  
8 which represents the character (see 15.2.10.1).

9 **15.2.20.5.9 IO#gets**

---

10 `gets`

---

11 **Visibility:** public

12 **Behavior:**

- 13 a) If the receiver is not readable, raise a direct instance of the class `IOError`.
- 14 b) If the receiver has reached its end, return `nil`.
- 15 c) Otherwise, read characters from the receiver until `0x0a` is read or the receiver reaches  
16 its end.
- 17 d) Create a direct instance of the class `String` whose content is the sequence of characters  
18 read in Step c) and return this instance.

19 **15.2.20.5.10 IO#initialize\_copy**

---

20 `initialize_copy(original)`

---

21 **Visibility:** private

22 **Behavior:** The behavior of the method is unspecified.

23 **15.2.20.5.11 IO#print**

---

24 `print(*args)`

---

1     **Visibility:** public

2     **Behavior:**

3     a) For each element of *args* in the indexing order:

4         1) Let *V* be the element. If the element is **nil**, a conforming processor may create  
5             a direct instance of the class **String** whose content is “nil” and let *V* be the  
6             instance.

7         2) Invoke the method **write** on the receiver with *V* as the argument.

8     b) Return an implementation-defined value.

9     **15.2.20.5.12 IO#putc**

---

10     **putc**( *obj* )

---

11     **Visibility:** public

12     **Behavior:**

13     a) If *obj* is not an instance of the class **Integer** or an instance of the class **String**, the  
14         behavior is unspecified. If *obj* is an instance of the class **Integer** whose value is smaller  
15         than 0 or larger than 255, the behavior is unspecified.

16     b) If *obj* is an instance of the class **Integer**, create a direct instance *S* of the class **String**  
17         whose content is a single character, whose character code is the value of *obj*.

18     c) If *obj* is an instance of the class **String**, create a direct instance *S* of the class **String**  
19         whose content is the first character of *obj*.

20     d) Invoke the method **write** on the receiver with *S* as the argument.

21     e) Return *obj*.

22     **15.2.20.5.13 IO#puts**

---

23     **puts**( *\*args* )

---

24     **Visibility:** public

25     **Behavior:**

26     a) If the length of *args* is 0, create a direct instance of the class **String** whose content is a  
27         single character 0x0a and invoke the method **write** on the receiver with this instance  
28         as an argument.

29     b) Otherwise, for each element *E* of *args* in the indexing order:

- 1) If  $E$  is an instance of the class `Array`, for each element  $X$  of  $E$  in the indexing order:
  - i) If  $X$  is the same object as  $E$ , i.e. if  $E$  contains itself, invoke the method `write` on the receiver with an instance of the class `String`, whose content is implementation-defined.
  - ii) Otherwise, invoke the method `write` on the receiver with  $X$  as the argument.
- 2) Otherwise:
  - i) If  $E$  is `nil`, a conforming processor may create a direct instance of the class `String` whose content is “nil” and let  $E$  be the instance.
  - ii) If  $E$  is not an instance of the class `String`, invoke the method `to_s` on the  $E$ . If the resulting value is an instance of the class `String`, let  $E$  be the resulting value. Otherwise, the behavior is unspecified.
  - iii) Invoke the method `write` on the receiver with  $E$  as the argument.
  - iv) If the last character of  $E$  is not `0x0a`, create a direct instance of the class `String` whose content is a single character `0x0a` and invoke the method `write` on the receiver with this instance as an argument.
- c) Return an implementation-defined value.

#### 15.2.20.5.14 IO#read

---

```
read( length=nil )
```

---

**Visibility:** public

**Behavior:**

- a) If the receiver is not readable, raise a direct instance of the class `IOError`.
- b) If the receiver has reached its end:
  - 1) If  $length$  is `nil`, create an empty instance of the class `String` and return that instance.
  - 2) If  $length$  is not `nil`, return `nil`.
- c) Otherwise:
  - 1) If  $length$  is `nil`, read characters from the receiver until the receiver reaches its end.
  - 2) If  $length$  is an instance of the class `Integer`, let  $N$  be the value of  $length$ . Otherwise, the behavior is unspecified.

- 1           3) If  $N$  is smaller than 0, raise a direct instance of the class `ArgumentError`.
- 2           4) Read bytes from the receiver until  $N$  bytes are read or the receiver reaches its end.
- 3         d) Create a direct instance of the class `String` whose content is the sequence of characters
- 4           read in Step c) and return this instance.

#### 5 **15.2.20.5.15 IO#readchar**

---

6         `readchar`

---

7         **Visibility:** public

8         **Behavior:**

- 9         a) If the receiver is not readable, raise a direct instance of the class `IOError`.
- 10        b) If the receiver has reached its end, raise a direct instance of the class `EOFError`.
- 11        c) Otherwise, read a character from the receiver. Return an instance of the class `Object`
- 12           which represents the character.

#### 13 **15.2.20.5.16 IO#readline**

---

14        `readline`

---

15        **Visibility:** public

16        **Behavior:**

- 17        a) If the receiver is not readable, raise a direct instance of the class `IOError`.
- 18        b) If the receiver has reached its end, raise a direct instance of the class `EOFError`.
- 19        c) Otherwise, read characters from the receiver until `0x0a` is read or the receiver reaches
- 20           its end.
- 21        d) Create a direct instance of the class `String` whose content is the sequence of characters
- 22           read in Step c) and return this instance.

#### 23 **15.2.20.5.17 IO#readlines**

---

24        `readlines`

---

25        **Visibility:** public

26        **Behavior:**

- 1 a) If the receiver is not readable, raise a direct instance of the class `IOError`.
- 2 b) Create an empty direct instance `A` of the class `Array`.
- 3 c) If the receiver has reached to its end, return `A`.
- 4 d) Otherwise, read characters from the receiver until `0x0a` is read or the receiver reaches  
5 its end.
- 6 e) Create a direct instance of the class `String` whose content is the sequence of characters  
7 read in Step d) and append this instance to `A`.
- 8 f) Continue processing from Step c).

#### 9 **15.2.20.5.18 IO#sync**

---

10 `sync`

---

11 **Visibility:** public

12 **Behavior:**

- 13 a) If the receiver is closed, raise a direct instance of the class `IOError`.
- 14 b) If the buffering flag of the receiver is true, return **false**. Otherwise, return **true**.

#### 15 **15.2.20.5.19 IO#sync=**

---

16 `sync=(bool)`

---

17 **Visibility:** public

18 **Behavior:**

- 19 a) If the receiver is closed, raise a direct instance of the class `IOError`.
- 20 b) If `bool` is a trueish object, set the buffering flag of the receiver to false. If `bool` is a  
21 falseish object, set the buffering flag of the receiver to true.
- 22 c) Return `bool`.

#### 23 **15.2.20.5.20 IO#write**

---

24 `write(str)`

---

25 **Visibility:** public

1     **Behavior:**

- 2     a) If *str* is an instance of the class `String`, let *S* be *str*.
- 3     b) Otherwise, invoke the method `to_s` on *str*, and let *S* be the resulting value. If *S* is not  
4       an instance of the class `String`, the behavior is unspecified.
- 5     c) If *S* is empty, return an instance of the class `Integer` whose value is 0.
- 6     d) If the receiver is not writable, raise a direct instance of the class `IOError`.
- 7     e) Write all the characters in *S* to the stream which the receiver represents, preserving  
8       their order.
- 9     f) Return an instance of the class `Integer`, whose value is implementation-defined.

10  **15.2.21 File**

11  **15.2.21.1 General description**

12  Instances of the class `File` represent opened files.

13  A conforming processor may raise an instance of the class `SystemCallError` during the execution  
14  of the methods of the class `File` if the underlying system reports an error.

15  An instance of the class `File` has the following attribute:

16     **path:** The sequence of characters which represents the location of the file. The correct  
17     syntax of paths is implementation-defined.

18  **15.2.21.2 Direct superclass**

19  The class `IO`

20  **15.2.21.3 Singleton methods**

21  **15.2.21.3.1 File.exist?**

---

22   `File.exist?( path )`

---

23   **Visibility:** public

24   **Behavior:**

- 25   a) If the file specified by *path* exists, return **true**.
- 26   b) Otherwise, return **false**.

27  **15.2.21.4 Instance methods**

28  **15.2.21.4.1 File#initialize**

---

1     `initialize( path, mode="r" )`

---

2     **Visibility:** private

3     **Behavior:**

- 4     a) If *path* is not an instance of the class **String**, the behavior is unspecified.
- 5     b) If *mode* is not an instance of the class **String** whose content is a single character “r”  
6         or “w”, the behavior is unspecified.
- 7     c) Open the file specified by *path* in an implementation-defined way, and associate it with  
8         the receiver.
- 9     d) Set the path of the receiver to the content of *path*.
- 10    e) Set the openness flag and the buffering flag of the receiver to true.
- 11    f) Set the readability flag and the writability flag of the receiver as follows:
- 12        1) If *mode* is an instance of the class **String** whose content is a single character “r”,  
13           set the readability flag of the receiver to true and set the writability flag of the  
14           receiver to false.
- 15        2) If *mode* is an instance of the class **String** whose content is a single character “w”,  
16           set the readability flag of the receiver to false and set the writability flag of the  
17           receiver to true.
- 18    g) Return an implementation-defined value.

#### 19    **15.2.21.4.2 File#path**

---

20     `path`

---

21     **Visibility:** public

22     **Behavior:** The method creates a direct instance of the class **String** whose content is the  
23     path of the receiver, and returns this instance.

#### 24    **15.2.22 Exception**

##### 25    **15.2.22.1 General description**

26    Instances of the class **Exception** represent exceptions. The class **Exception** is a superclass of  
27    all the other exception classes.

28    Instances of the class **Exception** have the following attribute.

29        **message:** An object returned by the method `to_s` (see 15.2.22.5.3).

1 When the method `clone` (see 15.3.1.3.8) or the method `dup` (see 15.3.1.3.9) of the class `Kernel`  
2 is invoked on an instance of the class `Exception`, the message attribute shall be copied from the  
3 receiver to the resulting value.

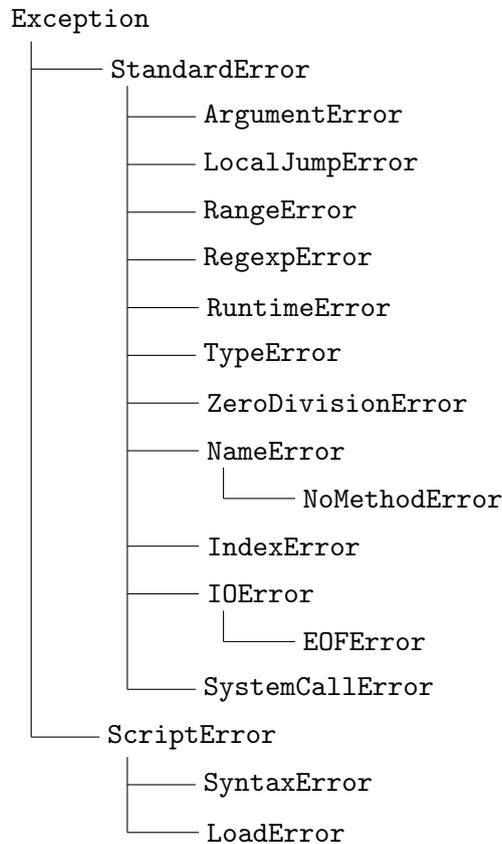
#### 4 **15.2.22.2 Direct superclass**

5 The class `Object`

#### 6 **15.2.22.3 Built-in exception classes**

7 This document defines several built-in subclasses of the class `Exception`. Figure 1 shows the  
8 list of these subclasses and their class hierarchy. Instances of these built-in subclasses are raised  
9 in various erroneous conditions as described in this document. The class hierarchy shown in  
10 Figure 1 is used to handle an exception (see Clause 14).

**Figure 1 – The exception class hierarchy**



#### 11 **15.2.22.4 Singleton methods**

##### 12 **15.2.22.4.1 Exception.exception**

---

13 `Exception.exception(*args, &block)`

---

14 **Visibility:** public

15 **Behavior:** Same as the method `new` (see 15.2.3.3.3).

1 **15.2.22.5 Instance methods**

2 **15.2.22.5.1 Exception#exception**

---

3 `exception(*string)`

---

4 **Visibility:** public

5 **Behavior:**

6 a) If the length of *string* is 0, return the receiver.

7 b) If the length of *string* is 1:

8 1) If the only argument is the same object as the receiver, return the receiver.

9 2) Otherwise let *M* be the argument.

10 i) Create a direct instance of the class of the receiver. Let *E* be the instance.

11 ii) Set the message attribute of *E* to *M*.

12 iii) Return *E*.

13 c) If the length of *string* is larger than 1, raise a direct instance of the class `ArgumentError`.

14 **15.2.22.5.2 Exception#message**

---

15 `message`

---

16 **Visibility:** public

17 **Behavior:**

18 a) Invoke the method `to_s` on the receiver with no arguments.

19 b) Return the resulting value of the invocation.

20 **15.2.22.5.3 Exception#to\_s**

---

21 `to_s`

---

22 **Visibility:** public

23 **Behavior:**

24 a) Let *M* be the message attribute of the receiver.

- 1       b) If *M* is **nil**, return an implementation-defined value.
- 2       c) If *M* is not an instance of the class **String**, the behavior is unspecified.
- 3       d) Otherwise, return *M*.

#### 4 **15.2.22.5.4 Exception#initialize**

---

5       **initialize**( *message=nil* )

---

6       **Visibility:** private

7       **Behavior:**

- 8       a) Set the message attribute of the receiver to *message*.
- 9       b) Return an implementation-defined value.

### 10 **15.2.23 StandardError**

#### 11 **15.2.23.1 General description**

12 Instances of the class **StandardError** represent standard errors, which can be handled in a  
13 *rescue-clause* without a *exception-class-list* (see 11.5.2.5).

#### 14 **15.2.23.2 Direct superclass**

15 The class **Exception**

### 16 **15.2.24 ArgumentError**

#### 17 **15.2.24.1 General description**

18 Instances of the class **ArgumentError** represent argument errors.

#### 19 **15.2.24.2 Direct superclass**

20 The class **StandardError**

### 21 **15.2.25 LocalJumpError**

22 Instances of the class **LocalJumpError** represent errors which occur while evaluating *blocks* and  
23 *jump-expressions*.

#### 24 **15.2.25.1 Direct superclass**

25 The class **StandardError**

#### 26 **15.2.25.2 Instance methods**

##### 27 **15.2.25.2.1 LocalJumpError#exit\_value**

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27

---

`exit_value`

---

**Visibility:** public

**Behavior:** The method returns the value of the instance variable `@exit_value` of the receiver.

#### 15.2.25.2.2 LocalJumpError#reason

---

`reason`

---

**Visibility:** public

**Behavior:** The method returns the value of the instance variable `@reason` of the receiver.

### 15.2.26 RangeError

#### 15.2.26.1 General description

Instances of the class `RangeError` represent range errors.

#### 15.2.26.2 Direct superclass

The class `StandardError`

### 15.2.27 RegexpError

#### 15.2.27.1 General description

Instances of the class `ArgumentError` represent regular expression errors.

#### 15.2.27.2 Direct superclass

The class `StandardError`

### 15.2.28 RuntimeError

#### 15.2.28.1 General description

Instances of the class `RuntimeError` represent runtime errors, which are raised by the method `raise` of the class `Kernel` by default (see 15.3.1.2.12).

#### 15.2.28.2 Direct superclass

The class `StandardError`

### 15.2.29 TypeError

#### 15.2.29.1 General description

Instances of the class `TypeError` represent type errors.

1 **15.2.29.2 Direct superclass**

2 The class `StandardError`

3 **15.2.30 ZeroDivisionError**

4 **15.2.30.1 General description**

5 Instances of the class `ZeroDivisionError` represent zero division errors.

6 **15.2.30.2 Direct superclass**

7 The class `StandardError`

8 **15.2.31 NameError**

9 Instances of the class `NameError` represent errors which occur while resolving names to values.

10 Instances of the class `NameError` have the following attribute.

11     **name:** The name a reference to which causes this exception to be raised.

12 When the method `clone` (see 15.3.1.3.8) or the method `dup` (see 15.3.1.3.9) of the class `Kernel`  
13 is invoked on an instance of the class `NameError`, the `name` attribute shall be copied from the  
14 receiver to the resulting value.

15 **15.2.31.1 Direct superclass**

16 The class `StandardError`

17 **15.2.31.2 Instance methods**

18 **15.2.31.2.1 NameError#name**

---

19     **name**

---

20     **Visibility:** public

21     **Behavior:** The method returns the `name` attribute of the receiver.

22 **15.2.31.2.2 NameError#initialize**

---

23     **initialize**( *message=nil, name=nil* )

---

24     **Visibility:** public

25     **Behavior:**

26     a) Set the `name` attribute of the receiver to the *name*.

- 1       b) Invoke the method `initialize` defined in the class `Exception`, which is a superclass of  
2       the class `NameError`, as if a *super-with-argument* were evaluated with a list of arguments  
3       which contains only *message* as the value of the *argument-without-parentheses* of the  
4       *super-with-argument*.
- 5       c) Return an implementation-defined value.

## 6 **15.2.32 NoMethodError**

7 Instances of the class `NoMethodError` represent errors which occur while invoking methods which  
8 do not exist or which cannot be invoked.

9 Instances of the class `NoMethodError` have attributes called ***name*** (see 15.2.31) and ***arguments***.  
10 The values of these attributes are set in the method `initialize` (see 15.2.32.2.2).

11 When the method `clone` (see 15.3.1.3.8) or the method `dup` (see 15.3.1.3.9) of the class `Kernel`  
12 is invoked on an instance of the class `NoMethodError`, those attributes shall be copied from the  
13 receiver to the resulting value.

### 14 **15.2.32.1 Direct superclass**

15 The class `NameError`

### 16 **15.2.32.2 Instance methods**

#### 17 **15.2.32.2.1 NoMethodError#args**

---

18       args

---

19       **Visibility:** public

20       **Behavior:** The method returns the value of the *arguments* attribute of the receiver.

#### 21 **15.2.32.2.2 NoMethodError#initialize**

---

22       initialize(*message=nil, name=nil, args=nil*)

---

23       **Visibility:** private

24       **Behavior:**

- 25       a) Set the *arguments* attribute of the receiver to the *args*.
- 26       b) Perform all the steps of the method `initialize` described in 15.2.31.2.2.
- 27       c) Return an implementation-defined value.

1 **15.2.33 IndexError**

2 **15.2.33.1 General description**

3 Instances of the class `IndexError` represent index errors.

4 **15.2.33.2 Direct superclass**

5 The class `StandardError`

6 **15.2.34 IOError**

7 **15.2.34.1 General description**

8 Instances of the class `IOError` represent input/output errors.

9 **15.2.34.2 Direct superclass**

10 The class `StandardError`

11 **15.2.35 EOFError**

12 **15.2.35.1 General description**

13 Instances of the class `EOFError` represent errors which occur when a stream has reached its end.

14 **15.2.35.2 Direct superclass**

15 The class `IOError`

16 **15.2.36 SystemCallError**

17 **15.2.36.1 General description**

18 Instances of the class `SystemCallError` represent errors which occur while invoking the methods  
19 of the class `IO`.

20 **15.2.36.2 Direct superclass**

21 The class `StandardError`

22 **15.2.37 ScriptError**

23 **15.2.37.1 General description**

24 Instances of the class `ScriptError` represent programming errors such as syntax errors and  
25 loading errors.

26 **15.2.37.2 Direct superclass**

27 The class `Exception`

1 **15.2.38 SyntaxError**

2 **15.2.38.1 General description**

3 Instances of the class `SyntaxError` represent syntax errors.

4 **15.2.38.2 Direct superclass**

5 The class `ScriptError`

6 **15.2.39 LoadError**

7 **15.2.39.1 General description**

8 Instances of the class `LoadError` represent errors which occur while loading external programs  
9 (see 15.3.1.2.13).

10 **15.2.39.2 Direct superclass**

11 The class `ScriptError`

12 **15.3 Built-in modules**

13 **15.3.1 Kernel**

14 **15.3.1.1 General description**

15 The module `Kernel` is included in the class `Object`. Unless overridden, instance methods defined  
16 in the module `Kernel` can be invoked on any instance of the class `Object`.

17 **15.3.1.2 Singleton methods**

18 **15.3.1.2.1 Kernel.‘**

---

19 `Kernel.‘ ( string )`

---

20 **Visibility:** public

21 **Behavior:** The method `‘` is invoked in the form described in 8.7.6.3.7.

22 The method `‘` executes an external command corresponding to *string*. The external com-  
23 mand executed by the method is implementation-defined.

24 When the method is invoked, take the following steps:

- 25 a) If *string* is not an instance of the class `String`, the behavior is unspecified.
- 26 b) Execute the command which corresponds to the content of *string*. Let *R* be the output  
27 of the command.
- 28 c) Create a direct instance of the class `String` whose content is *R*, and return the instance.

### 1 15.3.1.2.2 Kernel.block\_given?

---

2 Kernel.block\_given?

---

3 **Visibility:** public

4 **Behavior:**

- 5 a) If the top of `[[block]]` is block-not-given, return **false**.  
6 b) Otherwise, return **true**.

### 7 15.3.1.2.3 Kernel.eval

---

8 Kernel.eval(*string*)

---

9 **Visibility:** public

10 **Behavior:**

- 11 a) If *string* is not an instance of the class `String`, the behavior is unspecified.  
12 b) Parse the content of the *string* as a *program* (see 10.1). If it fails, raise a direct instance  
13 of the class `SyntaxError`.  
14 c) Evaluate the *program* (see 10.1) within the execution context as it exists just before  
15 this method invoked. Let *V* be the resulting value of the evaluation.  
16 d) Return *V*.

17 In Step c), the local variable scope which corresponds to the *program* is considered as a  
18 local variable scope which corresponds to a *block* in 9.2 d) 1).

19 EXAMPLE 1 The following program prints "123".

```
20 x = 123  
21 Kernel.eval("print x")
```

22 EXAMPLE 2 The following program raises an exception.

```
23 Kernel.eval("x = 123") # the scope of x is the program "x = 123".  
24 print x # x is undefined here.
```

### 25 15.3.1.2.4 Kernel.global\_variables

---

26 Kernel.global\_variables

---

27 **Visibility:** public

1     **Behavior:** The method returns a new direct instance of the class `Array` which consists of  
2     names of all the global variables. These names are represented by direct instances of either  
3     the class `String` or the class `Symbol`. Which of those classes is chosen is implementation-  
4     defined.

#### 5   15.3.1.2.5   Kernel.iterator?

---

6     `Kernel.iterator?`

---

7     **Visibility:** public

8     **Behavior:** Same as the method `Kernel.block_given?` (see 15.3.1.2.2).

#### 9   15.3.1.2.6   Kernel.lambda

---

10    `Kernel.lambda(&block)`

---

11    **Visibility:** public

12    **Behavior:** The method creates an instance of the class `Proc` as `Proc.new` does (see 15.2.17.3.1).  
13    However, the way in which *block* is evaluated differs from the one described in 11.3.3 except  
14    when *block* is called by a *yield-expression*.

15    The differences are as follows.

16    a) Before 11.3.3 d), the number of arguments is checked as follows:

17       1) Let  $A$  be the list of arguments passed to the block. Let  $N_a$  be the length of  $A$ .

18       2) If the *block-parameter-list* is of the form *left-hand-side*, and if  $N_a$  is not 1, the  
19       behavior is unspecified.

20       3) If the *block-parameter-list* is of the form *multiple-left-hand-side*:

21           i) If the *multiple-left-hand-side* is not of the form *grouped-left-hand-side* or *packing-*  
22           *left-hand-side*:

23               I) Let  $N_p$  be the number of *multiple-left-hand-side-items* of the *multiple-*  
24               *left-hand-side*.

25               II) If  $N_a < N_p$ , raise a direct instance of the class `ArgumentError`.

26               III) If a *packing-left-hand-side* is omitted, and if  $N_a > N_p$ , raise a direct  
27               instance of the class `ArgumentError`.

28           ii) If the *multiple-left-hand-side* is of the form *grouped-left-hand-side*, and if  $N_a$   
29           is not 1, the behavior is unspecified.

1 b) In 11.3.3 e), when the evaluation of the block associated with a `lambda` invocation is  
2 terminated by a *return-expression* or *break-expression*, the execution context is restored  
3 to  $E_o$  (i.e. the saved execution context), and the evaluation of the `lambda` invocation  
4 is terminated.

5 The value of the `lambda` invocation is determined as follows:

- 6 1) If the *jump-argument* of the *return-expression* or the *break-expression* is present,  
7 the value of the `lambda` invocation is the value of the *jump-argument*.
- 8 2) Otherwise, the value of the `lambda` invocation is **nil**.

#### 9 **15.3.1.2.7 Kernel.local\_variables**

---

10 `Kernel.local_variables`

---

11 **Visibility:** public

12 **Behavior:** The method returns a new direct instance of the class `Array` which contains all  
13 the names of local variable bindings which meet the following conditions.

- 14 • The name of the binding is of the form *local-variable-identifier*.
- 15 • The binding can be resolved in the scope of local variables which includes the point of  
16 invocations of this method by the process described in 9.2.

17 In the instance of the class `Array` returned by the method, names of the local variables are  
18 represented by instances of either the class `String` or the class `Symbol`. Which of those  
19 classes is chosen is implementation-defined.

#### 20 **15.3.1.2.8 Kernel.loop**

---

21 `Kernel.loop(&block)`

---

22 **Visibility:** public

23 **Behavior:**

- 24 a) If *block* is not given, the behavior is unspecified.
- 25 b) Otherwise, repeat calling *block*.

#### 26 **15.3.1.2.9 Kernel.p**

---

27 `Kernel.p(*args)`

---

28 **Visibility:** public

1     **Behavior:**

- 2     a) For each element *E* of *args*, in the indexing order, take the following steps:
- 3         1) Invoke the method `inspect` (see 15.3.1.3.17) on *E* with no arguments and let *X*  
4             be the resulting value of this invocation.
- 5         2) If *X* is not an instance of the class `String`, the behavior is unspecified.
- 6         3) Invoke the method `write`(see 15.2.20.5.20) on `Object::STDOUT` with *X* as the  
7             argument.
- 8         4) Invoke the method `write` on `Object::STDOUT` with an argument, which is a new  
9             direct instance of the class `String` whose content is a single character `0x0a`.
- 10     b) Return an implementation-defined value.

11 **15.3.1.2.10 Kernel.print**

---

12     `Kernel.print(*args)`

---

13     **Visibility:** public

14     **Behavior:** Invoke the method `print` of the class `IO` (see 15.2.20.5.11) on `Object::STDOUT`  
15     with the same arguments, and return the resulting value.

16 **15.3.1.2.11 Kernel.puts**

---

17     `Kernel.puts(*args)`

---

18     **Visibility:** public

19     **Behavior:** Invoke the method `puts` of the class `IO` (see 15.2.20.5.13) on `Object::STDOUT`  
20     with the same arguments, and return the resulting value.

21 **15.3.1.2.12 Kernel.raise**

---

22     `Kernel.raise(*args)`

---

23     **Visibility:** public

24     **Behavior:**

- 25     a) If the length of *args* is larger than 2, the behavior is unspecified.
- 26     b) If the length of *args* is 0:

- 1           1) If the location of the method invocation is within an *operator-expression*<sub>2</sub> of an  
2           *assignment-with-rescue-modifier*, a *fallback-statement-of-rescue-modifier-statement*,  
3           or a *rescue-clause*, let *E* be the current exception (see 14.3).
- 4           2) Otherwise, invoke the method `new` on the class `RuntimeError` with no argument.  
5           Let *E* be the resulting value.
- 6           c) If the length of *args* is 1, let *A* be the only argument.
  - 7           1) If *A* is an instance of the class `String`, invoke the method `new` on the class  
8           `RuntimeError` with *A* as the only argument. Let *E* be the resulting instance.
  - 9           2) Otherwise, invoke the method `exception` on *A*. Let *E* be the resulting value.
  - 10          3) If *E* is not an instance of the class `Exception`, raise a direct instance of the class  
11          `TypeError`.
- 12          d) If the length of *args* is 2, let *F* and *S* be the first and the second argument, respectively.
  - 13          1) Invoke the method `exception` on *F* with *S* as the only argument. Let *E* be the  
14          resulting value.
  - 15          2) If *E* is not an instance of the class `Exception`, raise a direct instance of the class  
16          `TypeError`.
- 17          e) Raise *E*.

### 18 15.3.1.2.13 Kernel.require

---

19 `Kernel.require( string )`

---

20 **Visibility:** public

21 **Behavior:** The method `require` evaluates the external program *P* corresponding to *string*.  
22 The way in which *P* is determined from *string* is implementation-defined.

23 When the method is invoked, take the following steps:

- 24           a) If *string* is not an instance of the class `String`, the behavior is unspecified.
- 25           b) Search for the external program *P* corresponding to *string*.
- 26           c) If the program does not exist, raise a direct instance of the class `LoadError`.
- 27           d) If *P* is not of the form *program* (see 10.1), raise a direct instance of the class `SyntaxError`.
- 28           e) Change the state of the execution context temporarily for the evaluation of *P* as follows:
  - 29           1) `[[self]]` contains only one object which is the object at the bottom of `[[self]]` in the  
30           current execution context.

- 1           2) `[[class-module-list]]` contains only one list whose only element is the class `Object`.
- 2           3) `[[default-method-visibility]]` contains only one visibility, which is the private visi-  
3           bility.
- 4           4) All the other attributes of the execution context are empty stacks.
- 5           f) Evaluate  $P$  within the execution context set up in Step e).
- 6           g) Restore the state of the execution context as it is just before Step e), even when an  
7           exception is raised and not handled during the evaluation of  $P$ .
- 8           NOTE The evaluation of  $P$  may affect the restored execution context if it changes the at-  
9           tributes of objects in the original execution context.
- 10          h) Unless an exception is raised and not handled in Step f), return **true**.

### 11 15.3.1.3 Instance methods

#### 12 15.3.1.3.1 `Kernel#==`

---

13           `==(other)`

---

14           **Visibility:** public

15           **Behavior:**

- 16          a) If the receiver and *other* are the same object, return **true**.
- 17          b) Otherwise, return **false**.

18 If the class `Object` is not the root of the class inheritance tree, the method `==` shall be defined  
19 in the class which is the root of the class inheritance tree instead of in the module `Kernel`.

#### 20 15.3.1.3.2 `Kernel#===`

---

21           `===(other)`

---

22           **Visibility:** public

23           **Behavior:**

- 24          a) If the receiver and *other* are the same object, return **true**.
- 25          b) Otherwise, invoke the method `==` on the receiver with *other* as the only argument. Let  
26           $V$  be the resulting value.
- 27          c) If  $V$  is a trueish object, return **true**. Otherwise, return **false**.

1 **15.3.1.3.3 Kernel#`__id__`**

---

2 `__id__`

---

3 **Visibility:** public

4 **Behavior:** Same as the method `object_id` (see 15.3.1.3.33).

5 **15.3.1.3.4 Kernel#`__send__`**

---

6 `__send__(symbol, *args, &block)`

---

7 **Visibility:** public

8 **Behavior:** Same as the method `send` (see 15.3.1.3.44).

9 If the class `Object` is not the root of the class inheritance tree, the method `__send__` shall be  
10 defined in the class which is the root of the class inheritance tree instead of in the module  
11 `Kernel`.

12 **15.3.1.3.5 Kernel#`'`**

---

13 `'(string)`

---

14 **Visibility:** private

15 **Behavior:** Same as the method `Kernel.'` (see 15.3.1.2.1).

16 **15.3.1.3.6 Kernel#`block_given?`**

---

17 `block_given?`

---

18 **Visibility:** private

19 **Behavior:** Same as the method `Kernel.block_given?` (see 15.3.1.2.2).

20 **15.3.1.3.7 Kernel#`class`**

---

21 `class`

---

22 **Visibility:** public

23 **Behavior:** The method returns the class of the receiver.

1 **15.3.1.3.8 Kernel#clone**

---

2 clone

---

3 **Visibility:** public

4 **Behavior:**

- 5 a) If the receiver is an instance of one of the following classes: NilClass, TrueClass,  
6 FalseClass, Integer, Float, or Symbol, then raise a direct instance of the class  
7 TypeError.
- 8 b) Create a direct instance of the class of the receiver which has no bindings of instance  
9 variables. Let  $O$  be the newly created instance.
- 10 c) For each binding  $B$  of the instance variables of the receiver, create a variable binding  
11 with the same name and value as  $B$  in the set of bindings of instance variables of  $O$ .
- 12 d) If the receiver is associated with a singleton class, let  $E_o$  be the singleton class, and  
13 take the following steps:
- 14 1) Create a singleton class whose direct superclass is the direct superclass of  $E_o$ . Let  
15  $E_n$  be the singleton class.
- 16 2) For each binding  $B_{v1}$  of the constants of  $E_o$ , create a variable binding with the  
17 same name and value as  $B_{v1}$  in the set of bindings of constants of  $E_n$ .
- 18 3) For each binding  $B_{v2}$  of the class variables of  $E_o$ , create a variable binding with  
19 the same name and value as  $B_{v2}$  in the set of bindings of class variables of  $E_n$ .
- 20 4) For each binding  $B_m$  of the instance methods of  $E_o$ , create a method binding with  
21 the same name and value as  $B_m$  in the set of bindings of instance methods of  $E_n$ .
- 22 5) Associate  $O$  with  $E_n$ .
- 23 e) Invoke the method `initialize_copy` on  $O$  with the receiver as the argument.
- 24 f) Return  $O$ .

25 **15.3.1.3.9 Kernel#dup**

---

26 dup

---

27 **Visibility:** public

28 **Behavior:**

- 29 a) If the receiver is an instance of one of the following classes: NilClass, TrueClass,  
30 FalseClass, Integer, Float, or Symbol, then raise a direct instance of the class  
31 TypeError.

- 1       b) Create a direct instance of the class of the receiver which has no bindings of instance  
2       variables. Let *O* be the newly created instance.
- 3       c) For each binding *B* of the instance variables of the receiver, create a variable binding  
4       with the same name and value as *B* in the set of bindings of instance variables of *O*.
- 5       d) Invoke the method `initialize_copy` on *O* with the receiver as the argument.
- 6       e) Return *O*.

7 **15.3.1.3.10 Kernel#eql?**

---

8       `eql?(other)`

---

9       **Visibility:** public

10       **Behavior:** Same as the method `==` (see 15.3.1.3.1).

11 **15.3.1.3.11 Kernel#equal?**

---

12       `equal?(other)`

---

13       **Visibility:** public

14       **Behavior:** Same as the method `==` (see 15.3.1.3.1).

15 If the class `Object` is not the root of the class inheritance tree, the method `equal?` shall be  
16 defined in the class which is the root of the class inheritance tree instead of in the module  
17 `Kernel`.

18 **15.3.1.3.12 Kernel#eval**

---

19       `eval(string)`

---

20       **Visibility:** private

21       **Behavior:** Same as the method `Kernel.eval` (see 15.3.1.2.3).

22 **15.3.1.3.13 Kernel#extend**

---

23       `extend(*module_list)`

---

24       **Visibility:** public

25       **Behavior:** Let *R* be the receiver of the method.

- 1 a) If the length of *module\_list* is 0, raise a direct instance of the class `ArgumentError`.
- 2 b) For each element *A* of *module\_list*, take the following steps:
  - 3 1) If *A* is not an instance of the class `Module`, raise a direct instance of the class
  - 4 `TypeError`.
  - 5 2) If *A* is an instance of the class `Class`, raise a direct instance of the class `TypeError`.
  - 6 3) Invoke the method `extend_object` on *A* with *R* as the only argument.
  - 7 4) Invoke the method `extended` on *A* with *R* as the only argument.
- 8 c) Return *R*.

#### 9 15.3.1.3.14 `Kernel#global_variables`

---

10 `global_variables`

---

11 **Visibility:** private

12 **Behavior:** Same as the method `Kernel.global_variables` (see 15.3.1.2.4).

#### 13 15.3.1.3.15 `Kernel#hash`

---

14 `hash`

---

15 **Visibility:** public

16 **Behavior:** The method returns an instance of the class `Integer`. When invoked on the  
17 same object, the method shall always return an instance of the class `Integer` whose value  
18 is the same.

19 When a conforming processor overrides the method `eq1?` (see 15.3.1.3.10), it shall override  
20 the method `hash` in the same class or module in which the method `eq1?` is overridden  
21 in such a way that, if an invocation of the method `eq1?` on an object with an argument  
22 returns a trueish object, invocations of the method `hash` on the object and the argument  
23 return the instances of the class `Integer` with the same value.

#### 24 15.3.1.3.16 `Kernel#initialize_copy`

---

25 `initialize_copy(original)`

---

26 **Visibility:** private

27 **Behavior:** The method `initialize_copy` is invoked when an object is created by the  
28 method `clone` (see 15.3.1.3.8) or the method `dup` (see 15.3.1.3.9).

29 When the method is invoked, take the following steps:

- 1 a) If the classes of the receiver and the *original* are not the same class, raise a direct  
2 instance of the class `TypeError`.
- 3 b) Return an implementation-defined value.

#### 4 **15.3.1.3.17 Kernel#inspect**

---

5 `inspect`

---

6 **Visibility:** public

7 **Behavior:** The method returns a new direct instance of the class `String`, the content of  
8 which represents the state of the receiver. The content of the resulting instance of the class  
9 `String` is implementation-defined.

#### 10 **15.3.1.3.18 Kernel#instance\_eval**

---

11 `instance_eval( string = nil, &block )`

---

12 **Visibility:** public

13 **Behavior:**

- 14 a) If the receiver is an instance of the class `Integer` or the class `Symbol`, or if the receiver  
15 is one of **nil**, **true**, or **false**, then the behavior is unspecified.
- 16 b) If the receiver is not associated with a singleton class, create a new singleton class. Let  
17 *M* be the newly created singleton class.
- 18 c) If the receiver is associated with a singleton class, let *M* be that singleton class.
- 19 d) Take steps b) through the last step of the method `class_eval` of the class `Module` (see  
20 15.2.2.4.15).

21 If the class `Object` is not the root of the class inheritance tree, the method `instance_eval` shall  
22 be defined in the class which is the root of the class inheritance tree instead of in the module  
23 `Kernel`.

#### 24 **15.3.1.3.19 Kernel#instance\_of?**

---

25 `instance_of?( module )`

---

26 **Visibility:** public

27 **Behavior:** Let *C* be the class of the receiver.

- 28 a) If *module* is not an instance of the class `Class` or the class `Module`, raise a direct  
29 instance of the class `TypeError`.

- 1        b) If *module* and *C* are the same object, return **true**.  
2        c) Otherwise, return **false**.

### 3 **15.3.1.3.20 Kernel#instance\_variable\_defined?**

---

4        `instance_variable_defined?(symbol)`

---

5        **Visibility:** public

6        **Behavior:**

- 7        a) Let *N* be the name designated by *symbol*.  
8        b) If *N* is not of the form *instance-variable-identifier*, raise a direct instance of the class  
9        `NameError` which has *symbol* as its name attribute.  
10       c) If a binding of an instance variable with name *N* exists in the set of bindings of instance  
11       variables of the receiver, return **true**.  
12       d) Otherwise, return **false**.

### 13 **15.3.1.3.21 Kernel#instance\_variable\_get**

---

14       `instance_variable_get(symbol)`

---

15       **Visibility:** public

16       **Behavior:**

- 17       a) Let *N* be the name designated by *symbol*.  
18       b) If *N* is not of the form *instance-variable-identifier*, raise a direct instance of the class  
19       `NameError` which has *symbol* as its name attribute.  
20       c) If a binding of an instance variable with name *N* exists in the set of bindings of instance  
21       variables of the receiver, return the value of the binding.  
22       d) Otherwise, return **nil**.

### 23 **15.3.1.3.22 Kernel#instance\_variable\_set**

---

24       `instance_variable_set(symbol, obj)`

---

25       **Visibility:** public

26       **Behavior:**

- 1 a) Let  $N$  be the name designated by *symbol*.
- 2 b) If  $N$  is not of the form *instance-variable-identifier*, raise a direct instance of the class  
3 `NameError` which has *symbol* as its name attribute.
- 4 c) If a binding of an instance variable with name  $N$  exists in the set of bindings of instance  
5 variables of the receiver, replace the value of the binding with *obj*.
- 6 d) Otherwise, create a variable binding with name  $N$  and value *obj* in the set of bindings  
7 of instance variables of the receiver.
- 8 e) Return *obj*.

#### 9 **15.3.1.3.23 Kernel#instance\_variables**

---

10 `instance_variables`

---

11 **Visibility:** public

12 **Behavior:** The method returns a new direct instance of the class `Array` which consists of  
13 names of all the instance variables of the receiver. These names are represented by direct  
14 instances of either the class `String` or the class `Symbol`. Which of those classes is chosen is  
15 implementation-defined.

#### 16 **15.3.1.3.24 Kernel#is\_a?**

---

17 `is_a?( module )`

---

18 **Visibility:** public

19 **Behavior:**

- 20 a) If *module* is not an instance of the class `Class` or the class `Module`, raise a direct  
21 instance of the class `TypeError`.
- 22 b) Let  $C$  be the class of the receiver.
- 23 c) If *module* is an instance of the class `Class` and one of the following conditions holds,  
24 return **true**.
  - 25 • The *module* and  $C$  are the same object.
  - 26 • The *module* is a superclass of  $C$ .
  - 27 • The *module* and the singleton class of the receiver are the same object.
- 28 d) If *module* is an instance of the class `Module` and is included in  $C$  or one of the super-  
29 classes of  $C$ , return **true**.
- 30 e) Otherwise, return **false**.

1 **15.3.1.3.25 Kernel#iterator?**

---

2 iterator?

---

3 **Visibility:** private

4 **Behavior:** Same as the method Kernel.iterator? (see 15.3.1.2.5).

5 **15.3.1.3.26 Kernel#kind\_of?**

---

6 kind\_of?( *module* )

---

7 **Visibility:** public

8 **Behavior:** Same as the method is\_a? (see 15.3.1.3.24).

9 **15.3.1.3.27 Kernel#lambda**

---

10 lambda( *&block* )

---

11 **Visibility:** private

12 **Behavior:** Same as the method Kernel.lambda (see 15.3.1.2.6).

13 **15.3.1.3.28 Kernel#local\_variables**

---

14 local\_variables

---

15 **Visibility:** private

16 **Behavior:** Same as the method Kernel.local\_variables (see 15.3.1.2.7).

17 **15.3.1.3.29 Kernel#loop**

---

18 loop( *&block* )

---

19 **Visibility:** private

20 **Behavior:** Same as the method Kernel.loop (see 15.3.1.2.8).

21 **15.3.1.3.30 Kernel#method\_missing**

---

1 `method_missing( symbol, *args )`

---

2 **Visibility:** private

3 **Behavior:**

4 a) If *symbol* is not an instance of the class `Symbol`, the behavior is unspecified.

5 b) Otherwise, raise a direct instance of the class `NoMethodError` which has *symbol* as  
6 its name attribute and *args* as its arguments attribute. A direct instance of the  
7 class `NameError` which has *symbol* as its name attribute may be raised instead of  
8 `NoMethodError` if the method is invoked in 13.3.3 e) during evaluation of a *local-*  
9 *variable-identifier* as a method invocation.

10 If the class `Object` is not the root of the class inheritance tree, the method `method_missing`  
11 shall be defined in the class which is the root of the class inheritance tree instead of in the  
12 module `Kernel`.

### 13 15.3.1.3.31 `Kernel#methods`

---

14 `methods( all=true )`

---

15 **Visibility:** public

16 **Behavior:** Let *C* be the class of the receiver.

17 a) If *all* is a trueish object, invoke the method `instance_methods` on *C* with no arguments  
18 (see 15.2.2.4.33), and return the resulting value.

19 b) If *all* is a falseish object, invoke the method `singleton_methods` on the receiver with  
20 **false** as the only argument (see 15.3.1.3.45), and return the resulting value.

### 21 15.3.1.3.32 `Kernel#nil?`

---

22 `nil?`

---

23 **Visibility:** public

24 **Behavior:**

25 a) If the receiver is **nil**, return **true**.

26 b) Otherwise, return **false**.

### 27 15.3.1.3.33 `Kernel#object_id`

---

1     `object_id`

---

2     **Visibility:** public

3     **Behavior:** The method returns an instance of the class `Integer` with the same value  
4     whenever it is invoked on the same object. When invoked on two distinct objects, the  
5     method returns an instance of the class `Integer` with different value for each invocation.

6     **15.3.1.3.34   Kernel#p**

---

7     `p(*args)`

---

8     **Visibility:** private

9     **Behavior:** Same as the method `Kernel.p` (see 15.3.1.2.9).

10    **15.3.1.3.35   Kernel#print**

---

11    `print(*args)`

---

12    **Visibility:** private

13    **Behavior:** Same as the method `Kernel.print` (see 15.3.1.2.10).

14    **15.3.1.3.36   Kernel#private\_methods**

---

15    `private_methods(all=true)`

---

16    **Visibility:** public

17    **Behavior:**

- 18    a) Let  $MV$  be the private visibility.
- 19    b) Create an empty direct instance  $A$  of the class `Array`.
- 20    c) If the receiver is associated with a singleton class, let  $C$  be the singleton class.
- 21    d) Let  $I$  be the set of bindings of instance methods of  $C$ .

22        For each binding  $B$  of  $I$ , let  $N$  and  $V$  be the name and the value of  $B$  respectively, and  
23        take the following steps:

- 24        1) If  $V$  is undef, or the visibility of  $V$  is not  $MV$ , skip the next two steps.

- 1           2) Let *S* be either a new direct instance of the class `String` whose content is *N* or a  
 2           direct instance of the class `Symbol` whose name is *N*. Which is chosen as the value  
 3           of *S* is implementation-defined.
- 4           3) Unless *A* contains the element of the same name (if *S* is an instance of the class  
 5           `Symbol`) or the same content (if *S* is an instance of the class `String`) as *S*, append  
 6           *S* to *A*.
- 7           e) For each module *M* in included module list of *C*, take step d), assuming that *C* in that  
 8           step to be *M*.
- 9           f) Let new *C* be the class of the receiver, and take step d).
- 10          g) If *all* is a trueish object:
- 11           1) Take step e).
- 12           2) If *C* does not have a direct superclass, return *A*.
- 13           3) Let new *C* be the direct superclass of current *C*.
- 14           4) Take step d), and then, repeat from Step g) 1).
- 15          h) Return *A*.

16 **15.3.1.3.37 Kernel#protected\_methods**

---

17 `protected_methods( all=true )`

---

18 **Visibility:** public

19 **Behavior:** Same as the method `private_methods` (see 15.3.1.3.36), except to let *MV* be  
 20 the protected visibility in 15.3.1.3.36 a).

21 **15.3.1.3.38 Kernel#public\_methods**

---

22 `public_methods( all=true )`

---

23 **Visibility:** public

24 **Behavior:** Same as the method `private_methods` (see 15.3.1.3.36), except to let *MV* be  
 25 the public visibility in 15.3.1.3.36 a).

26 **15.3.1.3.39 Kernel#puts**

---

1     `puts(*args)`

---

2     **Visibility:** private

3     **Behavior:** Same as the method `Kernel.puts` (see 15.3.1.2.11).

4     **15.3.1.3.40 Kernel#raise**

---

5     `raise(*args)`

---

6     **Visibility:** private

7     **Behavior:** Same as the method `Kernel.raise` (see 15.3.1.2.12).

8     **15.3.1.3.41 Kernel#remove\_instance\_variable**

---

9     `remove_instance_variable(symbol)`

---

10    **Visibility:** private

11    **Behavior:**

- 12    a) Let  $N$  be the name designated by *symbol*.
- 13    b) If  $N$  is not of the form *instance-variable-identifier*, raise a direct instance of the class  
14       **NameError** which has *symbol* as its name attribute.
- 15    c) If a binding of an instance variable with name  $N$  exists in the set of bindings of instance  
16       variables of the receiver, let  $V$  be the value of the binding.
- 17        1) Remove the binding from the set of bindings of instance variables of the receiver.
- 18        2) Return  $V$ .
- 19    d) Otherwise, raise a direct instance of the class **NameError** which has *symbol* as its name  
20       attribute.

21    **15.3.1.3.42 Kernel#require**

---

22    `require(*args)`

---

23    **Visibility:** private

24    **Behavior:** Same as the method `Kernel.require` (see 15.3.1.2.13).

1 **15.3.1.3.43 Kernel#respond\_to?**

---

2 `respond_to?( symbol, include_private=false )`

---

3 **Visibility:** public

4 **Behavior:**

- 5 a) Let *N* be the name designated by *symbol*.
- 6 b) Search for a binding of an instance method named *N* starting from the receiver of the  
7 method as described in 13.3.4.
- 8 c) If a binding is found, let *V* be the value of the binding.
- 9 1) If *V* is undef, return **false**.
- 10 2) If the visibility of *V* is private:
- 11 i) If *include\_private* is a trueish object, return **true**.
- 12 ii) Otherwise, return **false**.
- 13 3) Otherwise, return **true**.
- 14 d) Otherwise, return **false**.

15 **15.3.1.3.44 Kernel#send**

---

16 `send( symbol, *args, &block )`

---

17 **Visibility:** public

18 **Behavior:**

- 19 a) Let *N* be the name designated by *symbol*.
- 20 b) Invoke the method named *N* on the receiver with *args* as arguments and *block* as the  
21 block, if any.
- 22 c) Return the resulting value of the invocation.

23 **15.3.1.3.45 Kernel#singleton\_methods**

---

24 `singleton_methods( all=true )`

---

25 **Visibility:** public

- 1     **Behavior:** Let  $E$  be the singleton class of the receiver.
- 2     a) Create an empty direct instance  $A$  of the class `Array`.
- 3     b) Let  $I$  be the set of bindings of instance methods of  $E$ .
- 4         For each binding  $B$  of  $I$ , let  $N$  and  $V$  be the name and the value of  $B$  respectively, and
- 5         take the following steps:
- 6             1) If  $V$  is undef, or the visibility of  $V$  is private, skip the next two steps.
- 7             2) Let  $S$  be either a new direct instance of the class `String` whose content is  $N$  or a
- 8             direct instance of the class `Symbol` whose name is  $N$ . Which is chosen as the value
- 9             of  $S$  is implementation-defined.
- 10            3) Unless  $A$  contains the element of the same name (if  $S$  is an instance of the class
- 11            `Symbol`) or the same content (if  $S$  is an instance of the class `String`), append  $S$
- 12            to  $A$ .
- 13     c) If *all* is a trueish object, for each module  $M$  in included module list of  $E$ , take step b),
- 14     assuming that  $E$  in that step to be  $M$ .
- 15     d) Return  $A$ .

16 **15.3.1.3.46 Kernel#to\_s**

---

17     to\_s

---

18     **Visibility:** public

19     **Behavior:** The method returns a newly created direct instance of the class `String`, the

20     content of which is the string representation of the receiver. The content of the resulting

21     instance of the class `String` is implementation-defined.

22 **15.3.2 Enumerable**

23 **15.3.2.1 General description**

24 The module `Enumerable` provides methods which iterates over the elements of the object using

25 the method `each`.

26 In the following description of the methods of the module `Enumerable`, an *element* of the

27 receiver means one of the values which is yielded by the method `each`.

28 **15.3.2.2 Instance methods**

29 **15.3.2.2.1 Enumerable#all?**

---

1 `all?(&block)`

---

2 **Visibility:** public

3 **Behavior:**

4 a) Invoke the method **each** on the receiver.

5 b) For each element *X* which the method **each** yields:

6 1) If *block* is given, call *block* with *X* as the argument.

7 If this call results in a falseish object, return **false**.

8 2) If *block* is not given, and *X* is a falseish object, return **false**.

9 c) Return **true**.

10 **15.3.2.2.2 Enumerable#any?**

---

11 `any?(&block)`

---

12 **Visibility:** public

13 **Behavior:**

14 a) Invoke the method **each** on the receiver.

15 b) For each element *X* which **each** yields:

16 1) If *block* is given, call *block* with *X* as the argument.

17 If this call results in a trueish object, return **true**.

18 2) If *block* is not given, and *X* is a trueish object, return **true**.

19 c) Return **false**.

20 **15.3.2.2.3 Enumerable#collect**

---

21 `collect(&block)`

---

22 **Visibility:** public

23 **Behavior:**

24 a) If *block* is not given, the behavior is unspecified.

- 1 b) Create an empty direct instance *A* of the class `Array`.
- 2 c) Invoke the method `each` on the receiver.
- 3 d) For each element *X* which `each` yields, call *block* with *X* as the argument and append
- 4 the resulting value to *A*.
- 5 e) Return *A*.

#### 6 **15.3.2.2.4 Enumerable#detect**

---

```
7 detect(ifnone=nil, &block)
```

---

8 **Visibility:** public

9 **Behavior:**

- 10 a) If *block* is not given, the behavior is unspecified.
- 11 b) Invoke the method `each` on the receiver.
- 12 c) For each element *X* which `each` yields, call *block* with *X* as the argument. If this call
- 13 results in a trueish object, return *X*.
- 14 d) Return *ifnone*.

#### 15 **15.3.2.2.5 Enumerable#each\_with\_index**

---

```
16 each_with_index(&block)
```

---

17 **Visibility:** public

18 **Behavior:**

- 19 a) If *block* is not given, the behavior is unspecified.
- 20 b) Let *i* be 0.
- 21 c) Invoke the method `each` on the receiver.
- 22 d) For each element *X* which `each` yields:
  - 23 1) Call *block* with *X* and *i* as the arguments.
  - 24 2) Increase *i* by 1.
- 25 e) Return the receiver.

1 **15.3.2.2.6 Enumerable#entries**

---

2 `entries`

---

3 **Visibility:** public

4 **Behavior:**

- 5 a) Create an empty direct instance *A* of the class `Array`.
- 6 b) Invoke the method `each` on the receiver.
- 7 c) For each element *X* which `each` yields, append *X* to *A*.
- 8 d) Return *A*.

9 **15.3.2.2.7 Enumerable#find**

---

10 `find( ifnone=nil, &block)`

---

11 **Visibility:** public

12 **Behavior:** Same as the method `detect` (see 15.3.2.2.4).

13 **15.3.2.2.8 Enumerable#find\_all**

---

14 `find_all(&block)`

---

15 **Visibility:** public

16 **Behavior:**

- 17 a) If *block* is not given, the behavior is unspecified.
- 18 b) Create an empty direct instance *A* of the class `Array`.
- 19 c) Invoke the method `each` on the receiver.
- 20 d) For each element *X* which `each` yields, call *block* with *X* as the argument. If this call  
21 results in a trueish object, append *X* to *A*.
- 22 e) Return *A*.

23 **15.3.2.2.9 Enumerable#grep**

---

1 `grep( pattern, &block )`

---

2 **Visibility:** public

3 **Behavior:**

4 a) Create an empty direct instance *A* of the class **Array**.

5 b) Invoke the method **each** on the receiver.

6 c) For each element *X* which **each** yields, invoke the method **===** on *pattern* with *X* as  
7 the argument.

8 If this invocation results in a trueish object:

9 1) If *block* is given, call *block* with *X* as the argument and append the resulting value  
10 to *A*.

11 2) Otherwise, append *X* to *A*.

12 d) Return *A*.

13 **15.3.2.2.10 Enumerable#include?**

---

14 `include?( obj )`

---

15 **Visibility:** public

16 **Behavior:**

17 a) Invoke the method **each** on the receiver.

18 b) For each element *X* which **each** yields, invoke the method **==** on *X* with *obj* as the  
19 argument. If this invocation results in a trueish object, return **true**.

20 c) Return **false**.

21 **15.3.2.2.11 Enumerable#inject**

---

22 `inject( *args, &block )`

---

23 **Visibility:** public

24 **Behavior:**

25 a) If *block* is not given, the behavior is unspecified.

- 1 b) If the length of *args* is 2, the behavior is unspecified. If the length of *args* is larger  
2 than 2, raise a direct instance of the class `ArgumentError`.
- 3 c) Invoke the method `each` on the receiver. If the method `each` does not yield any element,  
4 return `nil`.
- 5 d) For each element *X* which `each` yields:
- 6 1) If *X* is the first element, and the length of *args* is 0, let *V* be *X*.
- 7 2) If *X* is the first element, and the length of *args* is 1, call *block* with two arguments,  
8 which are the only element of *args* and *X*. Let *V* be the resulting value of this call.
- 9 3) If *X* is not the first element, call *block* with *V* and *X* as the arguments. Let new  
10 *V* be the resulting value of this call.
- 11 e) Return *V*.

#### 12 15.3.2.2.12 Enumerable#map

---

13 `map(&block)`

---

14 **Visibility:** public

15 **Behavior:** Same as the method `collect` (see 15.3.2.2.3).

#### 16 15.3.2.2.13 Enumerable#max

---

17 `max(&block)`

---

18 **Visibility:** public

19 **Behavior:**

- 20 a) Invoke the method `each` on the receiver.
- 21 b) If the method `each` does not yield any elements, return `nil`.
- 22 c) For each element *X* which the method `each` yields:
- 23 1) If *X* is the first element, let *V* be *X*.
- 24 2) Otherwise:
- 25 i) If *block* is given:
- 26 I) Call *block* with *X* and *V* as the arguments. Let *D* be the result of this  
27 call.

- 1                   II) If  $D$  is not an instance of the class `Integer`, the behavior is unspecified.
- 2                   III) If the value of  $D$  is larger than 0, let new  $V$  be  $X$ .
- 3           ii) If *block* is not given:
- 4                   I) Invoke the method `<=>` on  $X$  with  $V$  as the argument. Let  $D$  be the
- 5                   result of this invocation.
- 6                   II) If  $D$  is not an instance of the class `Integer`, the behavior is unspecified.
- 7                   III) If the value of  $D$  is larger than 0, let new  $V$  be  $X$ .
- 8           d) Return  $V$ .

9 **15.3.2.2.14 Enumerable#min**

---

10 `min(&block)`

---

11 **Visibility:** public

12 **Behavior:**

- 13   a) Invoke the method `each` on the receiver.
- 14   b) If the method `each` does not yield any elements, return **nil**.
- 15   c) For each element  $X$  which the method `each` yields:
- 16       1) If  $X$  is the first element, let  $V$  be  $X$ .
- 17       2) Otherwise:
- 18           i) If *block* is given:
- 19                   I) Call *block* with  $X$  and  $V$  as the arguments. Let  $D$  be the result of this
- 20                   call.
- 21                   II) If  $D$  is not an instance of the class `Integer`, the behavior is unspecified.
- 22                   III) If the value of  $D$  is smaller than 0, let new  $V$  be  $X$ .
- 23           ii) If *block* is not given:
- 24                   I) Invoke the method `<=>` on  $X$  with  $V$  as the argument. Let  $D$  be the
- 25                   result of this invocation.
- 26                   II) If  $D$  is not an instance of the class `Integer`, the behavior is unspecified.
- 27                   III) If the value of  $D$  is smaller than 0, let new  $V$  be  $X$ .
- 28   d) Return  $V$ .

1 **15.3.2.2.15 Enumerable#member?**

---

2 `member?(obj)`

---

3 **Visibility:** public

4 **Behavior:** Same as the method `include?` (see 15.3.2.2.10).

5 **15.3.2.2.16 Enumerable#partition**

---

6 `partition(&block)`

---

7 **Visibility:** public

8 **Behavior:**

- 9 a) If *block* is not given, the behavior is unspecified.
- 10 b) Create two empty direct instances of the class `Array` *T* and *F*.
- 11 c) Invoke the method `each` on the receiver.
- 12 d) For each element *X* which `each` yields, call *block* with *X* as the argument.
- 13 If this call results in a trueish object, append *X* to *T*. If this call results in a falseish
- 14 object, append *X* to *F*.
- 15 e) Return a newly created an instance of the class `Array`, which contains only *T* and *F*
- 16 in this order.

17 **15.3.2.2.17 Enumerable#reject**

---

18 `reject(&block)`

---

19 **Visibility:** public

20 **Behavior:**

- 21 a) If *block* is not given, the behavior is unspecified.
- 22 b) Create an empty direct instance *A* of the class `Array`.
- 23 c) Invoke the method `each` on the receiver.
- 24 d) For each element *X* which `each` yields, call *block* with *X* as the argument. If this call
- 25 results in a falseish object, append *X* to *A*.
- 26 e) Return *A*.

1 **15.3.2.2.18 Enumerable#select**

---

2 `select(&block)`

---

3 **Visibility:** public

4 **Behavior:** Same as the method `find_all` (see 15.3.2.2.8).

5 **15.3.2.2.19 Enumerable#sort**

---

6 `sort(&block)`

---

7 **Visibility:** public

8 **Behavior:**

- 9 a) Create an empty direct instance *A* of the class `Array`.
- 10 b) Invoke the method `each` on the receiver.
- 11 c) Insert all the elements which the method `each` yields into *A*. For any two elements  $E_i$   
12 and  $E_j$  of *A*, the following condition shall hold:
- 13 1) Let *i* and *j* be the index of  $E_i$  and  $E_j$ , respectively.
- 14 2) If *block* is given:
- 15 i) Suppose *block* is called with  $E_i$  and  $E_j$  as the arguments.
- 16 ii) If this invocation does not result in an instance of the class `Integer`, the  
17 behavior is unspecified.
- 18 iii) If this invocation results in an instance of the class `Integer` whose value is  
19 larger than 0, *j* shall be larger than *i*.
- 20 iv) If this invocation results in an instance of the class `Integer` whose value is  
21 smaller than 0, *i* shall be larger than *j*.
- 22 3) If *block* is not given:
- 23 i) Suppose the method `<=>` is invoked on  $E_i$  with  $E_j$  as the argument.
- 24 ii) If this invocation does not result in an instance of the class `Integer`, the  
25 behavior is unspecified.
- 26 iii) If this invocation results in an instance of the class `Integer` whose value is  
27 larger than 0, *j* shall be larger than *i*.
- 28 iv) If this invocation results in an instance of the class `Integer` whose value is  
29 smaller than 0, *i* shall be larger than *j*.

1 d) Return *A*.

## 2 15.3.2.2.20 Enumerable#to\_a

---

3 to\_a

---

4 **Visibility:** public

5 **Behavior:** Same as the method `entries` (see 15.3.2.2.6).

## 6 15.3.3 Comparable

### 7 15.3.3.1 General description

8 The module `Comparable` provides methods which compare the receiver and an argument using  
9 the method `<=>`.

### 10 15.3.3.2 Instance methods

#### 11 15.3.3.2.1 Comparable#<

---

12 `<(other)`

---

13 **Visibility:** public

14 **Behavior:**

- 15 a) Invoke the method `<=>` on the receiver with *other* as the argument. Let *I* be the  
16 resulting value of this invocation.
- 17 b) If *I* is not an instance of the class `Integer`, the behavior is unspecified.
- 18 c) If the value of *I* is smaller than 0, return **true**. Otherwise, return **false**.

#### 19 15.3.3.2.2 Comparable#<=

---

20 `<=(other)`

---

21 **Visibility:** public

22 **Behavior:**

- 23 a) Invoke the method `<=>` on the receiver with *other* as the argument. Let *I* be the  
24 resulting value of this invocation.
- 25 b) If *I* is not an instance of the class `Integer`, the behavior is unspecified.
- 26 c) If the value of *I* is smaller than or equal to 0, return **true**. Otherwise, return **false**.

1 **15.3.3.2.3 Comparable#==**

---

2 ==(*other*)

---

3 **Visibility:** public

4 **Behavior:**

- 5 a) Invoke the method <=> on the receiver with *other* as the argument. Let *I* be the  
6 resulting value of this invocation.
- 7 b) If *I* is not an instance of the class `Integer`, the behavior is unspecified.
- 8 c) If the value of *I* is 0, return **true**. Otherwise, return **false**.

9 **15.3.3.2.4 Comparable#>**

---

10 >( *other* )

---

11 **Visibility:** public

12 **Behavior:**

- 13 a) Invoke the method <=> on the receiver with *other* as the argument. Let *I* be the  
14 resulting value of this invocation.
- 15 b) If *I* is not an instance of the class `Integer`, the behavior is unspecified.
- 16 c) If the value of *I* is larger than 0, return **true**. Otherwise, return **false**.

17 **15.3.3.2.5 Comparable#>=**

---

18 >=( *other* )

---

19 **Visibility:** public

20 **Behavior:**

- 21 a) Invoke the method <=> on the receiver with *other* as the argument. Let *I* be the  
22 resulting value of this invocation.
- 23 b) If *I* is not an instance of the class `Integer`, the behavior is unspecified.
- 24 c) If the value of *I* is larger than or equal to 0, return **true**. Otherwise, return **false**.

25 **15.3.3.2.6 Comparable#between?**

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11

---

`between?( left, right )`

---

**Visibility:** public

**Behavior:**

- a) Invoke the method `<=>` on the receiver with *left* as the argument. Let  $I_1$  be the resulting value of this invocation.
  - 1) If  $I_1$  is not an instance of the class `Integer`, the behavior is unspecified.
  - 2) If the value of  $I_1$  is smaller than 0, return **false**.
  
- b) Invoke the method `<=>` on the receiver with *right* as the argument. Let  $I_2$  be the resulting value of this invocation.
  - 1) If  $I_2$  is not an instance of the class `Integer`, the behavior is unspecified.
  - 2) If the value of  $I_2$  is larger than 0, return **false**. Otherwise, return **true**.