Home

Pages Classes Methods

Search

Table of Contents

What's Here
Querying
Instance Variables
Other

Show/hide navigation

Parent

BasicObject

Included Modules

Kernel

Methods

class Object

`Object` is the default root of all Ruby objects. `Object` inherits from `BasicObject` which allows creating alternate object hierarchies. Methods on `Object` are available to all classes unless explicitly overridden.

`Object` mixes in the `Kernel` module, making the built-in kernel functions globally accessible. Although the instance methods of `Object` are defined by the `Kernel` module, we have chosen to document them here for clarity.

When referencing constants in classes inheriting from `Object` you do not need to use the full namespace. For example, referencing `File` inside `YourClass` will find the top-level `File` class.

In the descriptions of Object's methods, the parameter *symbol* refers to a symbol, which is either a quoted string or a `Symbol` (such as `:name`).

What's Here

First, what's elsewhere. Class Object:

- Inherits from class BasicObject.
- Includes module Kernel.

Here, class Object provides methods for:

Querying

- #!~: Returns `true` if `self` does not match the given object, otherwise `false`.
- #<=>: Returns 0 if `self` and the given object `object` are the same object, or if `self == object`; otherwise returns `nil`.
- `===`: Implements case equality, effectively the same as calling `==`.
- `eql?`: Implements hash equality, effectively the same as calling `==`.
- `kind_of?` (aliased as `is_a?`): Returns whether given argument is an ancestor of the singleton class of `self`.
- `instance_of?`: Returns whether `self` is an instance of the given class.
- `instance_variable_defined?`: Returns whether the given instance variable is defined in `self`.
- `method`: Returns the `Method` object for the given method in `self`.
- `methods`: Returns an array of symbol names of public and protected methods in `self`.
- `nil?`: Returns `false`. (Only `nil` responds `true` to method `nil?`.)
- `object_id`: Returns an integer corresponding to `self` that is unique for the current process
- `private_methods`: Returns an array of the symbol names of the private methods in `self`.
- `protected_methods`: Returns an array of the symbol names of the protected methods in `self`.
- `public_method`: Returns the `Method` object for the given public method in `self`.
- `public_methods`: Returns an array of the symbol names of the public methods in `self`.
- `respond_to?`: Returns whether `self` responds to the given method.
- `singleton_class`: Returns the singleton class of `self`.
- `singleton_method`: Returns the `Method` object for the given singleton method in `self`.
- `singleton_methods`: Returns an array of the symbol names of the singleton methods in `self`.

- define_singleton_method : Defines a singleton method in `self` for the given symbol method-name and block or proc.
- extend : Includes the given modules in the singleton class of `self`.
- public_send : Calls the given public method in `self` with the given argument.
- send : Calls the given method in `self` with the given argument.

Instance Variables

- instance_variable_get : Returns the value of the given instance variable in `self`, or `nil` if the instance variable is not set.
- instance_variable_set : Sets the value of the given instance variable in `self` to the given object.
- instance_variables : Returns an array of the symbol names of the instance variables in `self`.
- remove_instance_variable : Removes the named instance variable from `self`.

Other

- clone : Returns a shallow copy of `self`, including singleton class and frozen state.
- define_singleton_method : Defines a singleton method in `self` for the given symbol method-name and block or proc.
- display : Prints `self` to the given IO stream or `$stdout`.
- dup : Returns a shallow unfrozen copy of `self`.
- enum_for (aliased as to_enum): Returns an Enumerator for `self` using the using the given method, arguments, and block.
- extend : Includes the given modules in the singleton class of `self`.
- freeze : Prevents further modifications to `self`.
- hash : Returns the integer hash value for `self`.
- inspect : Returns a human-readable string representation of `self`.
- itself : Returns `self`.
- method_missing : Method called when an undefined method is called on `self`.
- public_send : Calls the given public method in `self` with the given argument.
- send : Calls the given method in `self` with the given argument.

- to_s : Returns a string representation of `self`.

Constants

ARGF

ARGF is a stream designed for use in scripts that process files given as command-line arguments or passed in via STDIN.

See ARGF (the class) for more details.

ARGV

ARGV contains the command line arguments used to run ruby.

A library like OptionParser can be used to process command-line arguments.

DATA

DATA is a File that contains the data section of the executed file. To create a data section use `__END__`:

```
$ cat t.rb
puts DATA.gets
__END__
hello world!

$ ruby t.rb
hello world!
```

ENV

ENV is a Hash-like accessor for environment variables.

See ENV (the class) for more details.

RUBY_COPYRIGHT

The copyright string for ruby

RUBY_DESCRIPTION

The full ruby version string, like `ruby -v` prints

RUBY_ENGINE

The engine or interpreter this ruby uses.

RUBY_ENGINE_VERSION

The version of the engine or interpreter this ruby uses.

RUBY_PATCHLEVEL

The patchlevel for this ruby. If this is a development build of ruby the patchlevel will be -1

RUBY_PLATFORM

The platform for this ruby

RUBY_RELEASE_DATE

The date this ruby was released

RUBY_REVISION

The GIT commit hash for this ruby.

RUBY_VERSION

The running version of ruby

STDERR

Holds the original stderr

STDIN

Holds the original stdin

STDOUT

Holds the original stdout

TOPLEVEL_BINDING

The `Binding` of the top level scope

Public Instance Methods

obj !~ other → true or false

Returns true if two objects do not match (using the =~ method), otherwise false.

obj <=> other → 0 or nil

Returns 0 if `obj` and `other` are the same object or `obj == other`, otherwise nil.

The #<=> is used by various methods to compare objects, for example `Enumerable#sort`, `Enumerable#max` etc.

Your implementation of #<=> should return one of the following values: -1, 0, 1 or nil. -1 means self is smaller than other. 0 means self is equal to other. 1 means self is bigger than other. Nil means the two values could not be compared.

When you define #<=>, you can include `Comparable` to gain the methods #<=, #<, `==`, #>=, #> and between?.

true === other → true or false
false === other → true or false
nil === other → true or false

Returns `true` or `false`.

Like `Object#==`, if `object` is an instance of `Object` (and not an instance of one of its many subclasses).

This method is commonly overridden by those subclasses, to provide meaningful semantics in `case` statements.

define_singleton_method(symbol, method) → symbol
define_singleton_method(symbol) { block } → symbol

Defines a public singleton method in the receiver. The *method* parameter can be a `Proc`, a `Method` or an `UnboundMethod` object. If a block is specified, it is used as the method body. If a block or a method has parameters, they're used as method parameters.

```
class A
  class << self
    def class_name
      to_s
    end
  end
end
A.define_singleton_method(:who_am_i) do
  "I am: #{class_name}"
end
A.who_am_i   # ==> "I am: A"

guy = "Bob"
guy.define_singleton_method(:hello) { "#{self}: Hello there!" }
guy.hello    #=>  "Bob: Hello there!"

chris = "Chris"
```

```
chris.define_singleton_method(:greet) {|greeting| "#{greeting}, I'm Chris!" }
chris.greet("Hi") #=> "Hi, I'm Chris!"
```

display(port = $>) → nil

Writes `self` on the given port:

```
1.display
"cat".display
[ 4, 5, 6 ].display
puts
```

Output:

```
1cat[4, 5, 6]
```

dup → an_object

Produces a shallow copy of *obj*—the instance variables of *obj* are copied, but not the objects they reference.

This method may have class-specific behavior. If so, that behavior will be documented under the # `initialize_copy` method of the class.

on dup vs clone

In general, `clone` and `dup` may have different semantics in descendant classes. While `clone` is used to duplicate an object, including its internal state, `dup` typically uses the class of the descendant object to create the new instance.

When using `dup`, any modules that the object has been extended with will not be copied.

```
class Klass
  attr_accessor :str
end

module Foo
  def foo; 'foo'; end
end

s1 = Klass.new #=> #<Klass:0x401b3a38>
s1.extend(Foo) #=> #<Klass:0x401b3a38>
s1.foo #=> "foo"

s2 = s1.clone #=> #<Klass:0x401be280>
s2.foo #=> "foo"
```

```
s3 = s1.dup #=> #<Klass:0x401c1084>
s3.foo #=> NoMethodError: undefined method `foo' for #<Klass:0x401c1084>
```

enum_for(method = :each, *args) → enum
enum_for(method = :each, *args){|*args| block} → enum

Creates a new Enumerator which will enumerate by calling `method` on `obj`, passing `args` if any. What was *yielded* by method becomes values of enumerator.

If a block is given, it will be used to calculate the size of the enumerator without the need to iterate it (see Enumerator#size).

Examples

```
str = "xyz"

enum = str.enum_for(:each_byte)
enum.each { |b| puts b }
# => 120
# => 121
# => 122

# protect an array from being modified by some_method
a = [1, 2, 3]
some_method(a.to_enum)

# String#split in block form is more memory-effective:
very_large_string.split("|") { |chunk| return chunk if chunk.include?('DATE')
# This could be rewritten more idiomatically with to_enum:
very_large_string.to_enum(:split, "|").lazy.grep(/DATE/).first
```

It is typical to call to_enum when defining methods for a generic Enumerable, in case no block is passed.

Here is such an example, with parameter passing and a sizing block:

```
module Enumerable
  # a generic method to repeat the values of any enumerable
  def repeat(n)
    raise ArgumentError, "#{n} is negative!" if n < 0
    unless block_given?
      return to_enum(__method__, n) do # __method__ is :repeat here
        sz = size     # Call size and multiply by n...
        sz * n if sz  # but return nil if size itself is nil
      end
    end
    each do |*val|
      n.times { yield *val }
    end
  end
end

%i[hello world].repeat(2) { |w| puts w }
```

```
  # => Prints 'hello', 'hello', 'world', 'world'
enum = (1..14).repeat(3)
  # => returns an Enumerator when called without a block
enum.first(4) # => [1, 1, 1, 2]
enum.size # => 42
```

Alias for: to_enum

obj == other → true or false
equal?(other) → true or false
eql?(other) → true or false

Equality — At the Object level, == returns `true` only if `obj` and `other` are the same object. Typically, this method is overridden in descendant classes to provide class-specific meaning.

Unlike ==, the equal? method should never be overridden by subclasses as it is used to determine object identity (that is, `a.equal?(b)` if and only if `a` is the same object as `b`):

```
obj = "a"
other = obj.dup

obj == other      #=> true
obj.equal? other  #=> false
obj.equal? obj    #=> true
```

The eql? method returns `true` if `obj` and `other` refer to the same hash key. This is used by Hash to test members for equality. For any pair of objects where eql? returns `true`, the hash value of both objects must be equal. So any subclass that overrides eql? should also override hash appropriately.

For objects of class Object, eql? is synonymous with ==. Subclasses normally continue this tradition by aliasing eql? to their overridden == method, but there are exceptions. Numeric types, for example, perform type conversion across ==, but not across eql?, so:

```
1 == 1.0     #=> true
1.eql? 1.0   #=> false
```

extend(module, ...) → obj

Adds to *obj* the instance methods from each module given as a parameter.

```
module Mod
  def hello
    "Hello from Mod.\n"
  end
end
```

```
class Klass
  def hello
    "Hello from Klass.\n"
  end
end

k = Klass.new
k.hello         #=> "Hello from Klass.\n"
k.extend(Mod)   #=> #<Klass:0x401b3bc8>
k.hello         #=> "Hello from Mod.\n"
```

freeze → obj

Prevents further modifications to *obj*. A FrozenError will be raised if modification is attempted. There is no way to unfreeze a frozen object. See also Object#frozen?.

This method returns self.

```
a = [ "a", "b", "c" ]
a.freeze
a << "z"
```

produces:

```
prog.rb:3:in `<<': can't modify frozen Array (FrozenError)
 from prog.rb:3
```

Objects of the following classes are always frozen: Integer, Float, Symbol.

hash → integer

Generates an Integer hash value for this object. This function must have the property that `a.eql?(b)` implies `a.hash == b.hash`.

The hash value is used along with eql? by the Hash class to determine if two objects reference the same hash key. Any hash value that exceeds the capacity of an Integer will be truncated before being used.

The hash value for an object may not be identical across invocations or implementations of Ruby. If you need a stable identifier across Ruby invocations and implementations you will need to generate one with a custom method.

Certain core classes such as Integer use built-in hash calculations and do not call the hash method when used as a hash key.

When implementing your own hash based on multiple values, the best practice is to combine the class and any values using the hash code of an array:

For example:

```
def hash
  [self.class, a, b, c].hash
end
```

The reason for this is that the Array#hash method already has logic for safely and efficiently combining multiple hash values.

inspect → string

Returns a string containing a human-readable representation of *obj*. The default inspect shows the object's class name, an encoding of its memory address, and a list of the instance variables and their values (by calling inspect on each of them). User defined classes should override this method to provide a better representation of *obj*. When overriding this method, it should return a string whose encoding is compatible with the default external encoding.

```
[ 1, 2, 3..4, 'five' ].inspect   #=> "[1, 2, 3..4, \"five\"]"
Time.new.inspect                 #=> "2008-03-08 19:43:39 +0900"

class Foo
end
Foo.new.inspect                  #=> "#<Foo:0x0300c868>"

class Bar
  def initialize
    @bar = 1
  end
end
Bar.new.inspect                  #=> "#<Bar:0x0300c868 @bar=1>"
```

instance_of?(class) → true or false

Returns `true` if *obj* is an instance of the given class. See also Object#kind_of?.

```
class A;     end
class B < A; end
class C < B; end

b = B.new
b.instance_of? A   #=> false
b.instance_of? B   #=> true
b.instance_of? C   #=> false
```

instance_variable_defined?(symbol) → true or false
instance_variable_defined?(string) → true or false

Returns `true` if the given instance variable is defined in *obj*. String arguments are converted to symbols.

```
class Fred
  def initialize(p1, p2)
    @a, @b = p1, p2
  end
end
fred = Fred.new('cat', 99)
fred.instance_variable_defined?(:@a)    #=> true
fred.instance_variable_defined?("@b")   #=> true
fred.instance_variable_defined?("@c")   #=> false
```

instance_variable_get(symbol) → obj
instance_variable_get(string) → obj

Returns the value of the given instance variable, or nil if the instance variable is not set. The `@` part of the variable name should be included for regular instance variables. Throws a `NameError` exception if the supplied symbol is not valid as an instance variable name. `String` arguments are converted to symbols.

```
class Fred
  def initialize(p1, p2)
    @a, @b = p1, p2
  end
end
fred = Fred.new('cat', 99)
fred.instance_variable_get(:@a)    #=> "cat"
fred.instance_variable_get("@b")   #=> 99
```

instance_variable_set(symbol, obj) → obj
instance_variable_set(string, obj) → obj

Sets the instance variable named by *symbol* to the given object. This may circumvent the encapsulation intended by the author of the class, so it should be used with care. The variable does not have to exist prior to this call. If the instance variable name is passed as a string, that string is converted to a symbol.

```
class Fred
  def initialize(p1, p2)
    @a, @b = p1, p2
  end
end
fred = Fred.new('cat', 99)
fred.instance_variable_set(:@a, 'dog')   #=> "dog"
fred.instance_variable_set(:@c, 'cat')   #=> "cat"
fred.inspect                             #=> "#<Fred:0x401b3da8 @a=\"dog\", @
```

instance_variables → array

Returns an array of instance variable names for the receiver. Note that simply defining an accessor does not create the corresponding instance variable.

```
class Fred
  attr_accessor :a1
  def initialize
    @iv = 3
  end
end
Fred.new.instance_variables   #=> [:@iv]
```

is_a?(class) → true or false

Returns `true` if *class* is the class of *obj*, or if *class* is one of the superclasses of *obj* or modules included in *obj*.

```
module M;    end
class A
  include M
end
class B < A; end
class C < B; end

b = B.new
b.is_a? A          #=> true
b.is_a? B          #=> true
b.is_a? C          #=> false
b.is_a? M          #=> true

b.kind_of? A       #=> true
b.kind_of? B       #=> true
b.kind_of? C       #=> false
b.kind_of? M       #=> true
```

Alias for: *kind_of?*

itself → obj

Returns the receiver.

```
string = "my string"
string.itself.object_id == string.object_id   #=> true
```

kind_of?(class) → true or false

Returns `true` if *class* is the class of *obj*, or if *class* is one of the superclasses of *obj* or modules included in *obj*.

```
module M;    end
class A
  include M
end
class B < A; end
class C < B; end

b = B.new
b.is_a? A          #=> true
b.is_a? B          #=> true
b.is_a? C          #=> false
b.is_a? M          #=> true

b.kind_of? A       #=> true
b.kind_of? B       #=> true
b.kind_of? C       #=> false
b.kind_of? M       #=> true
```

Also aliased as: is_a?

method(sym) → method

Looks up the named method as a receiver in *obj*, returning a Method object (or raising NameError). The Method object acts as a closure in *obj*'s object instance, so instance variables and the value of `self` remain available.

```
class Demo
  def initialize(n)
    @iv = n
  end
  def hello()
    "Hello, @iv = #{@iv}"
  end
end

k = Demo.new(99)
m = k.method(:hello)
m.call   #=> "Hello, @iv = 99"

l = Demo.new('Fred')
m = l.method("hello")
m.call   #=> "Hello, @iv = Fred"
```

Note that Method implements `to_proc` method, which means it can be used with iterators.

```
[ 1, 2, 3 ].each(&method(:puts)) # => prints 3 lines to stdout

out = File.open('test.txt', 'w')
[ 1, 2, 3 ].each(&out.method(:puts)) # => prints 3 lines to file

require 'date'
%w[2017-03-01 2017-03-02].collect(&Date.method(:parse))
#=> [#<Date: 2017-03-01 ((2457814j,0s,0n),+0s,2299161j)>, #<Date: 2017-03-02
```

methods(regular=true) → array

Returns a list of the names of public and protected methods of *obj*. This will include all the methods accessible in *obj*'s ancestors. If the optional parameter is `false`, it returns an array of *obj*'s public and protected singleton methods, the array will not include methods in modules included in *obj*.

```
class Klass
  def klass_method()
  end
end
k = Klass.new
k.methods[0..9]    #=> [:klass_method, :nil?, :===,
                   #     :==~, :!, :eql?
                   #     :hash, :<=>, :class, :singleton_class]
k.methods.length   #=> 56

k.methods(false)   #=> []
def k.singleton_method; end
k.methods(false)   #=> [:singleton_method]

module M123; def m123; end end
k.extend M123
k.methods(false)   #=> [:singleton_method]
```

nil? → true or false

Only the object *nil* responds `true` to `nil?`.

```
Object.new.nil?   #=> false
nil.nil?          #=> true
```

__id__ → integer
object_id → integer

Returns an integer identifier for `obj`.

The same number will be returned on all calls to `object_id` for a given object, and no two active objects will share an id.

Note: that some objects of builtin classes are reused for optimization. This is the case for immediate values and frozen string literals.

`BasicObject` implements +__id__+, `Kernel` implements `object_id`.

Immediate values are not passed by reference but are passed by value: `nil`, `true`, `false`, Fixnums, Symbols, and some Floats.

```
Object.new.object_id  == Object.new.object_id  # => false
(21 * 2).object_id    == (21 * 2).object_id    # => true
"hello".object_id     == "hello".object_id     # => false
"hi".freeze.object_id == "hi".freeze.object_id # => true
```

private_methods(all=true) → array

Returns the list of private methods accessible to *obj*. If the *all* parameter is set to `false`, only those methods in the receiver will be listed.

protected_methods(all=true) → array

Returns the list of protected methods accessible to *obj*. If the *all* parameter is set to `false`, only those methods in the receiver will be listed.

public_method(sym) → method

Similar to *method*, searches public method only.

public_methods(all=true) → array

Returns the list of public methods accessible to *obj*. If the *all* parameter is set to `false`, only those methods in the receiver will be listed.

public_send(symbol [, args...]) → obj
public_send(string [, args...]) → obj

Invokes the method identified by *symbol*, passing it any arguments specified. Unlike send, `public_send` calls public methods only. When the method is identified by a string, the string is converted to a symbol.

```
1.public_send(:puts, "hello")  # causes NoMethodError
```

remove_instance_variable(symbol) → obj
remove_instance_variable(string) → obj

Removes the named instance variable from *obj*, returning that variable's value. `String` arguments are converted to symbols.

```
class Dummy
  attr_reader :var
```

```
  def initialize
    @var = 99
  end
  def remove
    remove_instance_variable(:@var)
  end
end
d = Dummy.new
d.var      #=> 99
d.remove   #=> 99
d.var      #=> nil
```

respond_to?(symbol, include_all=false) → true or false
respond_to?(string, include_all=false) → true or false

Returns `true` if *obj* responds to the given method. Private and protected methods are included in the search only if the optional second parameter evaluates to `true`.

If the method is not implemented, as `Process.fork` on Windows, `File.lchmod` on GNU/Linux, etc., false is returned.

If the method is not defined, `respond_to_missing?` method is called and the result is returned.

When the method name parameter is given as a string, the string is converted to a symbol.

respond_to_missing?(symbol, include_all) → true or false
respond_to_missing?(string, include_all) → true or false

DO NOT USE THIS DIRECTLY.

Hook method to return whether the *obj* can respond to *id* method or not.

When the method name parameter is given as a string, the string is converted to a symbol.

See `respond_to?`, and the example of `BasicObject`.

send(symbol [, args...]) → obj
__send__(symbol [, args...]) → obj
send(string [, args...]) → obj
__send__(string [, args...]) → obj

Invokes the method identified by *symbol*, passing it any arguments specified. When the method is identified by a string, the string is converted to a symbol.

`BasicObject` implements +__send__+, `Kernel` implements `send`. `__send__` is safer than `send` when *obj* has the same method name like `Socket`. See also `public_send`.

```
class Klass
  def hello(*args)
    "Hello " + args.join(' ')
  end
end
k = Klass.new
k.send :hello, "gentle", "readers"   #=> "Hello gentle readers"
```

singleton_class → class

Returns the singleton class of *obj*. This method creates a new singleton class if *obj* does not have one.

If *obj* is `nil`, `true`, or `false`, it returns NilClass, TrueClass, or FalseClass, respectively. If *obj* is an Integer, a Float or a Symbol, it raises a TypeError.

```
Object.new.singleton_class  #=> #<Class:#<Object:0xb7ce1e24>>
String.singleton_class      #=> #<Class:String>
nil.singleton_class         #=> NilClass
```

singleton_method(sym) → method

Similar to *method*, searches singleton method only.

```
class Demo
  def initialize(n)
    @iv = n
  end
  def hello()
    "Hello, @iv = #{@iv}"
  end
end

k = Demo.new(99)
def k.hi
  "Hi, @iv = #{@iv}"
end
m = k.singleton_method(:hi)
m.call   #=> "Hi, @iv = 99"
m = k.singleton_method(:hello) #=> NameError
```

singleton_methods(all=true) → array

Returns an array of the names of singleton methods for *obj*. If the optional *all* parameter is true, the list will include methods in modules included in *obj*. Only public and protected singleton methods are returned.

```
module Other
  def three() end
```

```
end

class Single
  def Single.four() end
end

a = Single.new

def a.one()
end

class << a
  include Other
  def two()
  end
end

Single.singleton_methods    #=> [:four]
a.singleton_methods(false)  #=> [:two, :one]
a.singleton_methods         #=> [:two, :one, :three]
```

to_enum(method = :each, *args) → enum
to_enum(method = :each, *args) {|*args| block} → enum

Creates a new Enumerator which will enumerate by calling `method` on `obj`, passing `args` if any. What was *yielded* by method becomes values of enumerator.

If a block is given, it will be used to calculate the size of the enumerator without the need to iterate it (see Enumerator#size).

Examples

```
str = "xyz"

enum = str.enum_for(:each_byte)
enum.each { |b| puts b }
# => 120
# => 121
# => 122

# protect an array from being modified by some_method
a = [1, 2, 3]
some_method(a.to_enum)

# String#split in block form is more memory-effective:
very_large_string.split("|") { |chunk| return chunk if chunk.include?('DATE')
# This could be rewritten more idiomatically with to_enum:
very_large_string.to_enum(:split, "|").lazy.grep(/DATE/).first
```

It is typical to call to_enum when defining methods for a generic Enumerable, in case no block is passed.

Here is such an example, with parameter passing and a sizing block:

```
module Enumerable
  # a generic method to repeat the values of any enumerable
  def repeat(n)
    raise ArgumentError, "#{n} is negative!" if n < 0
    unless block_given?
      return to_enum(__method__, n) do # __method__ is :repeat here
        sz = size     # Call size and multiply by n...
        sz * n if sz  # but return nil if size itself is nil
      end
    end
    each do |*val|
      n.times { yield *val }
    end
  end
end

%i[hello world].repeat(2) { |w| puts w }
  # => Prints 'hello', 'hello', 'world', 'world'
enum = (1..14).repeat(3)
  # => returns an Enumerator when called without a block
enum.first(4) # => [1, 1, 1, 2]
enum.size # => 42
```

Also aliased as: enum_for

to_s → string

Returns a string representing *obj*. The default `to_s` prints the object's class and an encoding of the object id. As a special case, the top-level object that is the initial execution context of Ruby programs returns "main".

Validate
Generated by RDoc 6.4.0.
Based on Darkfish by Michael Granger.
Ruby-doc.org is a service of James Britt and Neurogami, purveyors of fine dance noise