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).

In Files

```
class.c
enumerator.c
eval.c
gc.c
hash.c
io.c
object.c
parse.c
proc.c
ruby.c
version.c
vm.c
vm_eval.c
vm_method.c
Parent
BasicObject
```

Included Modules

```
Kernel
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.

```
FALSE
```

An alias of false

```
NIL_P
```

An alias of nil

```
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 SVN revision for this ruby.

```
RUBY_VERSION
```

The running version of ruby

```
SCRIPT_LINES__
```

When a Hash is assigned to SCRIPT_LINES__ the contents of files loaded after the assignment will be added as an Array of lines with the file name as the key.

```
STDERR
```

Holds the original stderr

```
STDIN
```

Holds the original stdin

```
STDOUT
```

Holds the original stdout

```
TOPLEVEL_BINDING
```

The Binding of the top level scope

```
TRUE
```

An alias of true

Public Instance Methods

obj !~ other → true or false

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

```
static VALUE
rb_obj_not_match(VALUE obj1, VALUE obj2)
{
    VALUE result = rb_funcall(obj1, id_match, 1, obj2);
    return RTEST(result) ? Qfalse : Qtrue;
}
```

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?.

```
static VALUE
rb_obj_cmp(VALUE obj1, VALUE obj2)
{
    if (obj1 == obj2 || rb_equal(obj1, obj2))
        return INT2FIX(0);
    return Qnil;
}
```

obj === other → true or false

Case Equality – For class Object, effectively the same as calling #==, but typically overridden by descendants to provide meaningful semantics in case statements.

```
VALUE
rb_equal(VALUE obj1, VALUE obj2)
{
    VALUE result;

    if (obj1 == obj2) return Qtrue;
    result = rb_funcall(obj1, id_eq, 1, obj2);
    if (RTEST(result)) return Qtrue;
    return Qfalse;
```

```
}
```

obj =~ other → nil

Pattern Match—Overridden by descendants (notably Regexp and String) to provide meaningful pattern-match semantics.

```
static VALUE
rb_obj_match(VALUE obj1, VALUE obj2)
{
    return Qnil;
}
```

class → class

Returns the class of obj. This method must always be called with an explicit receiver, as class is also a reserved word in Ruby.

```
1.class      #=> Fixnum
self.class   #=> Object
```

```
VALUE
rb_obj_class(VALUE obj)
{
    return rb_class_real(CLASS_OF(obj));
}
```

clone → an_object

Produces a shallow copy of obj—the instance variables of obj are copied, but not the objects they reference. clone copies the frozen and tainted state of obj. See also the discussion under Object#dup.

```
class Klass
   attr_accessor :str
end
s1 = Klass.new      #=> #<Klass:0x401b3a38>
s1.str = "Hello"    #=> "Hello"
s2 = s1.clone       #=> #<Klass:0x401b3998 @str="Hello">
s2.str[1,4] = "i"   #=> "i"
s1.inspect          #=> "#<Klass:0x401b3a38 @str=\"Hi\">"
s2.inspect          #=> "#<Klass:0x401b3998 @str=\"Hi\">"
```

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

```
VALUE
```

```
rb_obj_clone(VALUE obj)
{
    VALUE clone;
    VALUE singleton;

    if (rb_special_const_p(obj)) {
        rb_raise(rb_eTypeError, "can't clone %s", rb_obj_classname(obj));
    }
    clone = rb_obj_alloc(rb_obj_class(obj));
    RBASIC(clone)->flags &= (FL_TAINT|FL_PROMOTED0|FL_PROMOTED1);
    RBASIC(clone)->flags |= RBASIC(obj)->flags & ~
(FL_PROMOTED0|FL_PROMOTED1|FL_FREEZE|FL_FINALIZE);

    singleton = rb_singleton_class_clone_and_attach(obj, clone);
    RBASIC_SET_CLASS(clone, singleton);
    if (FL_TEST(singleton, FL_SINGLETON)) {
        rb_singleton_class_attached(singleton, clone);
    }

    init_copy(clone, obj);
    rb_funcall(clone, id_init_clone, 1, obj);
    RBASIC(clone)->flags |= RBASIC(obj)->flags & FL_FREEZE;

    return clone;
}
```

define_singleton_method(symbol, method) → new_method

define_singleton_method(symbol) { block } → proc

Defines a 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.

```
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!"
```

```
static VALUE
rb_obj_define_method(int argc, VALUE *argv, VALUE obj)
{
    VALUE klass = rb_singleton_class(obj);

    return rb_mod_define_method(argc, argv, klass);
}
```

display(port=$>) → nil

Prints obj on the given port (default $>). Equivalent to:

```
def display(port=$>)
  port.write self
end
For example:

1.display
"cat".display
[ 4, 5, 6 ].display
putsproduces:


1cat456
```

```
static VALUE
rb_obj_display(int argc, VALUE *argv, VALUE self)
{
    VALUE out;

    if (argc == 0) {
        out = rb_stdout;
    }
    else {
        rb_scan_args(argc, argv, "01", &out);
    }
    rb_io_write(out, self);

    return Qnil;
}
```

dup → an_object

Produces a shallow copy of obj—the instance variables of obj are copied, but not the objects they reference. dup copies the tainted state of obj.

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:0x401b3a38>
s2.foo #=> "foo"

s3 = s1.dup #=> #<Klass:0x401b3a38>
s3.foo #=> NoMethodError: undefined method `foo' for #<Klass:0x401b3a38>
```

```
VALUE
rb_obj_dup(VALUE obj)
{
    VALUE dup;

    if (rb_special_const_p(obj)) {
        rb_raise(rb_eTypeError, "can't dup %s", rb_obj_classname(obj));
    }
    dup = rb_obj_alloc(rb_obj_class(obj));
    init_copy(dup, obj);
    rb_funcall(dup, id_init_dup, 1, obj);
```

```
    return dup;
}
```

to_enum(method = :each, *args) → enum

enum_for(method = :each, *args) → enum

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

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)
```

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

%[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
```

```
static VALUE
obj_to_enum(int argc, VALUE *argv, VALUE obj)
{
    VALUE enumerator, meth = sym_each;

    if (argc > 0) {
        --argc;
        meth = *argv++;
    }
    enumerator = rb_enumeratorize_with_size(obj, meth, argc, argv, 0);
    if (rb_block_given_p()) {
        enumerator_ptr(enumerator)->size = rb_block_proc();
    }
    return enumerator;
}
```

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 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
```

```
VALUE
rb_obj_equal(VALUE obj1, VALUE obj2)
{
    if (obj1 == obj2) return Qtrue;
    return Qfalse;
}
```

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"
```

```
static VALUE
rb_obj_extend(int argc, VALUE *argv, VALUE obj)
{
    int i;
    ID id_extend_object, id_extended;

    CONST_ID(id_extend_object, "extend_object");
    CONST_ID(id_extended, "extended");
```

```
    rb_check_arity(argc, 1, UNLIMITED_ARGUMENTS);
    for (i = 0; i < argc; i++)
        Check_Type(argv[i], T_MODULE);
    while (argc--) {
        rb_funcall(argv[argc], id_extend_object, 1, obj);
        rb_funcall(argv[argc], id_extended, 1, obj);
    }
    return obj;
}
```

freeze → obj

Prevents further modifications to obj. A RuntimeError 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 (RuntimeError)
 from prog.rb:3
```

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

```
VALUE
rb_obj_freeze(VALUE obj)
{
    if (!OBJ_FROZEN(obj)) {
        OBJ_FREEZE(obj);
        if (SPECIAL_CONST_P(obj)) {
            rb_bug("special consts should be frozen.");
        }
    }
    return obj;
}
```

frozen? → true or false

Returns the freeze status of obj.

```
a = [ "a", "b", "c" ]
a.freeze    #=> ["a", "b", "c"]
a.frozen?   #=> true
```

```
VALUE
rb_obj_frozen_p(VALUE obj)
{
    return OBJ_FROZEN(obj) ? Qtrue : Qfalse;
}
```

hash → fixnum

Generates a Fixnum 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 a Fixnum 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.

```
VALUE
rb_obj_hash(VALUE obj)
{
    VALUE oid = rb_obj_id(obj);
#if SIZEOF_LONG == SIZEOF_VOIDP
    st_index_t index = NUM2LONG(oid);
#elif SIZEOF_LONG_LONG == SIZEOF_VOIDP
    st_index_t index = NUM2LL(oid);
#else
# error not supported
#endif
    return LONG2FIX(rb_objid_hash(index));
}
```

inspect → string

Returns a string containing a human-readable representation of obj. The default inspect shows the object's class name, an encoding of the object id, 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>"
```

```
static VALUE
rb_obj_inspect(VALUE obj)
{
    if (rb_ivar_count(obj) > 0) {
        VALUE str;
        VALUE c = rb_class_name(CLASS_OF(obj));

        str = rb_sprintf("-<%"PRIsVALUE":%p", c, (void*)obj);
        return rb_exec_recursive(inspect_obj, obj, str);
    }
    else {
        return rb_any_to_s(obj);
    }
}
```

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
```

```
VALUE
rb_obj_is_instance_of(VALUE obj, VALUE c)
{
    c = class_or_module_required(c);
    if (rb_obj_class(obj) == c) return Qtrue;
    return Qfalse;
}
```

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
```

```
static VALUE
rb_obj_ivar_defined(VALUE obj, VALUE iv)
{
    ID id = id_for_var(obj, iv, an, instance);

    if (!id) {
        return Qfalse;
    }
    return rb_ivar_defined(obj, id);
}
```

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
```

```
static VALUE
rb_obj_ivar_get(VALUE obj, VALUE iv)
```

```
{
    ID id = id_for_var(obj, iv, an, instance);

    if (!id) {
        return Qnil;
    }
    return rb_ivar_get(obj, id);
}
```

instance_variable_set(symbol, obj) → obj

instance_variable_set(string, obj) → obj

Sets the instance variable named by symbol to the given object, thereby frustrating the efforts of the class's author to attempt to provide proper encapsulation. 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\", @b=99,
@c=\"cat\">"
```

```
static VALUE
rb_obj_ivar_set(VALUE obj, VALUE iv, VALUE val)
{
    ID id = id_for_var(obj, iv, an, instance);
    if (!id) id = rb_intern_str(iv);
    return rb_ivar_set(obj, id, val);
}
```

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]
```

```
VALUE
rb_obj_instance_variables(VALUE obj)
{
    VALUE ary;

    ary = rb_ary_new();
    rb_ivar_foreach(obj, ivar_i, ary);
    return ary;
}
```

is_a?(class) → true or false

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
```

```
VALUE
rb_obj_is_kind_of(VALUE obj, VALUE c)
{
    VALUE cl = CLASS_OF(obj);

    c = class_or_module_required(c);
    return class_search_ancestor(cl, RCLASS_ORIGIN(c)) ? Qtrue : Qfalse;
}
```

itself → an_object

Returns obj.

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

```
static VALUE
rb_obj_itself(VALUE obj)
{
    return obj;
}
```

is_a?(class) → true or false

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
```

```
VALUE
rb_obj_is_kind_of(VALUE obj, VALUE c)
{
    VALUE cl = CLASS_OF(obj);

    c = class_or_module_required(c);
    return class_search_ancestor(cl, RCLASS_ORIGIN(c)) ? Qtrue : Qfalse;
}
```

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"
```

```
VALUE
rb_obj_method(VALUE obj, VALUE vid)
{
    return obj_method(obj, vid, FALSE);
}
```

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]
```

```
VALUE
rb_obj_methods(int argc, const VALUE *argv, VALUE obj)
{
    rb_check_arity(argc, 0, 1);
    if (argc > 0 && !RTEST(argv[0])) {
    return rb_obj_singleton_methods(argc, argv, obj);
    }
    return class_instance_method_list(argc, argv, CLASS_OF(obj), 1, ins_methods_i);
}
```

nil? → true or false

Only the object nil responds true to nil?.

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

```
static VALUE
rb_false(VALUE obj)
{
    return Qfalse;
}
```

__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.

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
```

```
VALUE
rb_obj_id(VALUE obj)
{
    /*
     *                32-bit VALUE space
     *          MSB ------------------------ LSB
     *  false   00000000000000000000000000000000
     *  true    00000000000000000000000000000010
     *  nil     00000000000000000000000000000100
     *  undef   00000000000000000000000000000110
     *  symbol  ssssssssssssssssssssssss00001110
     *  object  oooooooooooooooooooooooooooooo00        = 0 (mod sizeof(RVALUE))
     *  fixnum  ffffffffffffffffffffffffffffffff1
     *
     *                    object_id space
     *                                       LSB
     *  false   00000000000000000000000000000000
     *  true    00000000000000000000000000000010
     *  nil     00000000000000000000000000000100
     *  undef   00000000000000000000000000000110
     *  symbol   000SSSSSSSSSSSSSSSSSSSSSSSSSSS0        S...S % A = 4 (S...S = s...s
* A + 4)
     *  object   oooooooooooooooooooooooooooooo0        o...o % A = 0
     *  fixnum  ffffffffffffffffffffffffffffffff1       bignum if required
     *
     *  where A = sizeof(RVALUE)/4
     *
     *  sizeof(RVALUE) is
     *  20 if 32-bit, double is 4-byte aligned
     *  24 if 32-bit, double is 8-byte aligned
     *  40 if 64-bit
     */
    if (STATIC_SYM_P(obj)) {
        return (SYM2ID(obj) * sizeof(RVALUE) + (4 << 2)) | FIXNUM_FLAG;
    }
    else if (FLONUM_P(obj)) {
#if SIZEOF_LONG == SIZEOF_VOIDP
        return LONG2NUM((SIGNED_VALUE)obj);
#else
```

```
        return LL2NUM((SIGNED_VALUE)obj);
#endif
    }
    else if (SPECIAL_CONST_P(obj)) {
        return LONG2NUM((SIGNED_VALUE)obj);
    }
    return nonspecial_obj_id(obj);
}
```

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.

```
VALUE
rb_obj_private_methods(int argc, const VALUE *argv, VALUE obj)
{
    return class_instance_method_list(argc, argv, CLASS_OF(obj), 1,
ins_methods_priv_i);
}
```

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.

```
VALUE
rb_obj_protected_methods(int argc, const VALUE *argv, VALUE obj)
{
    return class_instance_method_list(argc, argv, CLASS_OF(obj), 1,
ins_methods_prot_i);
}
```

public_method(sym) → method

Similar to method, searches public method only.

```
VALUE
rb_obj_public_method(VALUE obj, VALUE vid)
{
    return obj_method(obj, vid, TRUE);
}
```

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.

```
VALUE
rb_obj_public_methods(int argc, const VALUE *argv, VALUE obj)
{
    return class_instance_method_list(argc, argv, CLASS_OF(obj), 1,
ins_methods_pub_i);
}
```

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
```

```
VALUE
rb_f_public_send(int argc, VALUE *argv, VALUE recv)
{
    return send_internal(argc, argv, recv, CALL_PUBLIC);
}
```

remove_instance_variable(symbol) → obj

Removes the named instance variable from obj, returning that variable's value.

```
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
```

```
VALUE
rb_obj_remove_instance_variable(VALUE obj, VALUE name)
{
```

```
    VALUE val = Qnil;
    const ID id = id_for_var(obj, name, an, instance);
    st_data_t n, v;
    struct st_table *iv_index_tbl;
    st_data_t index;

    rb_check_frozen(obj);
    if (!id) {
    goto not_defined;
    }

    switch (BUILTIN_TYPE(obj)) {
      case T_OBJECT:
        iv_index_tbl = ROBJECT_IV_INDEX_TBL(obj);
        if (!iv_index_tbl) break;
        if (!st_lookup(iv_index_tbl, (st_data_t)id, &index)) break;
        if (ROBJECT_NUMIV(obj) <= (long)index) break;
        val = ROBJECT_IVPTR(obj)[index];
        if (val != Qundef) {
            ROBJECT_IVPTR(obj)[index] = Qundef;
            return val;
        }
    break;
      case T_CLASS:
      case T_MODULE:
    n = id;
    if (RCLASS_IV_TBL(obj) && st_delete(RCLASS_IV_TBL(obj), &n, &v)) {
        return (VALUE)v;
    }
    break;
      default:
    if (FL_TEST(obj, FL_EXIVAR)) {
        if (generic_ivar_remove(obj, id, &val)) {
        return val;
        }
    }
    break;
    }

  not_defined:
    rb_name_err_raise("instance variable %1$s not defined, obj, name);
    UNREACHABLE;
}
```

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.

```
static VALUE
obj_respond_to(int argc, VALUE *argv, VALUE obj)
{
    VALUE mid, priv;
    ID id;
    rb_thread_t *th = GET_THREAD();

    rb_scan_args(argc, argv, "11", &mid, &priv);
    if (!(id = rb_check_id(&mid))) {
        VALUE ret = basic_obj_respond_to_missing(th, CLASS_OF(obj), obj,
                                                 rb_to_symbol(mid), priv);
        if (ret == Qundef) ret = Qfalse;
        return ret;
    }
    if (basic_obj_respond_to(th, obj, id, !RTEST(priv)))
        return Qtrue;
    return Qfalse;
}
```

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.

```
static VALUE
obj_respond_to_missing(VALUE obj, VALUE mid, VALUE priv)
{
    return Qfalse;
}
```

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. You can use **send** if the name send clashes with an existing method in obj. When the method is identified by a string, the string is converted to a symbol.

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

```
VALUE
rb_f_send(int argc, VALUE *argv, VALUE recv)
{
    return send_internal(argc, argv, recv, CALL_FCALL);
}
```

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 a Fixnum or a Symbol, it raises a TypeError.

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

```
static VALUE
rb_obj_singleton_class(VALUE obj)
{
    return rb_singleton_class(obj);
}
```

singleton_method(sym) → method

Similar to method, searches singleton method only.

```ruby
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
```

```
VALUE
rb_obj_singleton_method(VALUE obj, VALUE vid)
{
    const rb_method_entry_t *me;
    VALUE klass;
    ID id = rb_check_id(&vid);

    if (!id) {
        if (!NIL_P(klass = rb_singleton_class_get(obj)) &&
            respond_to_missing_p(klass, obj, vid, FALSE)) {
            id = rb_intern_str(vid);
            return mnew_missing(klass, obj, id, id, rb_cMethod);
        }
      undef:
        rb_name_err_raise("undefined singleton method `%1$s' for `%2$s'",
                          obj, vid);
    }
    if (NIL_P(klass = rb_singleton_class_get(obj)) ||
        UNDEFINED_METHOD_ENTRY_P(me = rb_method_entry_at(klass, id)) ||
        UNDEFINED_REFINED_METHOD_P(me->def)) {
        vid = ID2SYM(id);
        goto undef;
    }
    return mnew_from_me(me, klass, obj, id, rb_cMethod, FALSE);
}
```

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]
```

```
VALUE
rb_obj_singleton_methods(int argc, const VALUE *argv, VALUE obj)
{
    VALUE recur, ary, klass, origin;
    struct method_entry_arg me_arg;
    struct rb_id_table *mtbl;

    if (argc == 0) {
    recur = Qtrue;
    }
    else {
    rb_scan_args(argc, argv, "01", &recur);
    }
    klass = CLASS_OF(obj);
    origin = RCLASS_ORIGIN(klass);
    me_arg.list = st_init_numtable();
    me_arg.recur = RTEST(recur);
    if (klass && FL_TEST(klass, FL_SINGLETON)) {
```

```
    if ((mtbl = RCLASS_M_TBL(origin)) != 0) rb_id_table_foreach(mtbl,
method_entry_i, &me_arg);
    klass = RCLASS_SUPER(klass);
    }
    if (RTEST(recur)) {
    while (klass && (FL_TEST(klass, FL_SINGLETON) || RB_TYPE_P(klass, T_ICLASS))) {
        if (klass != origin && (mtbl = RCLASS_M_TBL(klass)) != 0)
rb_id_table_foreach(mtbl, method_entry_i, &me_arg);
        klass = RCLASS_SUPER(klass);
    }
    }
    ary = rb_ary_new();
    st_foreach(me_arg.list, ins_methods_i, ary);
    st_free_table(me_arg.list);

    return ary;
}
```

taint → obj

Mark the object as tainted.

Objects that are marked as tainted will be restricted from various built-in methods. This is to prevent insecure data, such as command-line arguments or strings read from Kernel#gets, from inadvertently compromising the user's system.

To check whether an object is tainted, use tainted?.

You should only untaint a tainted object if your code has inspected it and determined that it is safe. To do so use untaint.

```
VALUE
rb_obj_taint(VALUE obj)
{
    if (!OBJ_TAINTED(obj) && OBJ_TAINTABLE(obj)) {
        rb_check_frozen(obj);
        OBJ_TAINT(obj);
    }
    return obj;
}
```

tainted? → true or false

Returns true if the object is tainted.

See taint for more information.

```
VALUE
rb_obj_tainted(VALUE obj)
{
    if (OBJ_TAINTED(obj))
        return Qtrue;
    return Qfalse;
}
```

tap{|x|...} → obj

Yields self to the block, and then returns self. The primary purpose of this method is to "tap into" a method chain, in order to perform operations on intermediate results within the chain.

```
(1..10)               .tap {|x| puts "original: #{x.inspect}"}
  .to_a               .tap {|x| puts "array: #{x.inspect}"}
  .select {|x| x%2==0} .tap {|x| puts "evens: #{x.inspect}"}
  .map {|x| x*x}       .tap {|x| puts "squares: #{x.inspect}"}
```

```
VALUE
rb_obj_tap(VALUE obj)
{
    rb_yield(obj);
    return obj;
}
```

to_enum(method = :each, *args) → enum

enum_for(method = :each, *args) → enum

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

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)
```

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

%[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
```

```
static VALUE
obj_to_enum(int argc, VALUE *argv, VALUE obj)
{
    VALUE enumerator, meth = sym_each;

    if (argc > 0) {
        --argc;
        meth = *argv++;
    }
    enumerator = rb_enumeratorize_with_size(obj, meth, argc, argv, 0);
    if (rb_block_given_p()) {
        enumerator_ptr(enumerator)->size = rb_block_proc();
    }
```

```
    return enumerator;
}
```

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''.

```
VALUE
rb_any_to_s(VALUE obj)
{
    VALUE str;
    VALUE cname = rb_class_name(CLASS_OF(obj));

    str = rb_sprintf("#<%"PRIsVALUE":%p>", cname, (void*)obj);
    OBJ_INFECT(str, obj);

    return str;
}
```

trust → obj

Deprecated method that is equivalent to untaint.

```
VALUE
rb_obj_trust(VALUE obj)
{
    rb_warning("trust is deprecated and its behavior is same as untaint");
    return rb_obj_untaint(obj);
}
```

untaint → obj

Removes the tainted mark from the object.

See taint for more information.

```
VALUE
rb_obj_untaint(VALUE obj)
{
    if (OBJ_TAINTED(obj)) {
        rb_check_frozen(obj);
        FL_UNSET(obj, FL_TAINT);
    }
    return obj;
}
```

untrust → obj

Deprecated method that is equivalent to taint.

```
VALUE
rb_obj_untrust(VALUE obj)
{
    rb_warning("untrust is deprecated and its behavior is same as taint");
    return rb_obj_taint(obj);
}
```

untrusted? → true or false

Deprecated method that is equivalent to tainted?.

```
VALUE
rb_obj_untrusted(VALUE obj)
{
    rb_warning("untrusted? is deprecated and its behavior is same as tainted?");
    return rb_obj_tainted(obj);
}
```

Origianally Generated by RDoc 3.12.3.

Origianally Generated with the Darkfish Rdoc Generator 3.

Copied to a mardown file and converted to pdf.

https://code.visualstudio.com/c