Class

Object

Subclasses: Array, Binding, Continuation, Data (used internally by the interpreter), Dir, Exception, FalseClass, File::Stat, Hash, IO, MatchData, Method, Module, NilClass, Numeric, Proc, Process::Status, Range, Regexp, String, Struct, Symbol, Thread, ThreadGroup, Time, TrueClass, UnboundMethod

Object is the parent class of all classes in Ruby. Its methods are therefore available to all objects 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.

In the descriptions that follow, the parameter *symbol* refers to a symbol, which is either a quoted string or a Symbol (such as :name).

Instance methods

===

obj === *other_obj* → true or false

Case Equality—A synonym for Object#== but typically overridden by descendents to provide meaningful semantics in case statements.

=~

obj =~ *other_obj* → nil

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

!~

obj =~ *other_obj* → !(*obj*=~ *other_obj*)

1.9

Opposite of =~.

class

obj.class → *klass*

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

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

clone

obj.clone → *other_obj*

Produces a shallow copy of *obj*—the instance variables of *obj* are copied, but not the objects they reference. 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:0x0a2f1c>
s1.str = "Hello"    # =>   "Hello"
s2 = s1.clone       # =>   #<Klass:0x0a2cb0 @str="Hello">
s2.str[1,4] = "i"   # =>   "i"
s1.inspect          # =>   "#<Klass:0x0a2f1c @str=\"Hi\">"
s2.inspect          # =>   "#<Klass:0x0a2cb0 @str=\"Hi\">"
```

define_singleton_method

obj.define_singleton_method(*symbol*, *method*) → *method*
obj.define_method(*symbol*) { *block* } → *proc*

1.9 Defines a singleton method in the receiver. The *method* parameter can be a Proc or Method object. If a block is specified, it is used as the method body. This block is evaluated using instance_eval. See also Module#define_method.

```
a = "cat"
a.define_singleton_method(:speak) do
  puts "miaow"
end
a.speak
```

produces:

```
miaow
```

define_singleton_method is also useful with Module#class_eval:

```
class Test
end
Test.class_eval do
  define_method(:one) { puts "instance method" }
  define_singleton_method(:two) { puts "class method" }
end
t = Test.new
t.one
Test.two
```

produces:

```
instance method
class method
```

display

obj.display(*port*=$>) → nil

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

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

For example:

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

produces:

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

dup

obj.dup → *other_obj*

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*. See also the discussion under Object#clone. In general, clone and dup may have different semantics in descendent classes. Although clone is used to duplicate an object, including its internal state, dup typically uses the class of the descendent object to create the new instance.

enum_for

obj.enum_for(*using*=:each, ⟨ args ⟩$^+$ → *enumerator*

1.9

Synonym for Object#to_enum.

eql?

obj.eql?(*other_obj*) → true or false

Returns true if *obj* and *other_obj* have the same value. Used by Hash to test members for equality. For objects of class Object, eql? is synonymous with ==. Subclasses normally continue this tradition, but there are exceptions. Numeric types, for example, perform type conversion across ==, but not across eql?. This means that

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

extend

obj.extend(⟨ *mod* ⟩$^+$) → *obj*

Adds to *obj* the instance methods from each module given as a parameter. See also Module#extend_object.

```
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:0x0a3200>
k.hello          # =>   "Hello from Mod.\n"
```

Writing *obj*.extend(Mod) is basically the same as the following:

```
class <<obj
  include Mod
end
```

freeze

obj.freeze → *obj*

Prevents further modifications to *obj*. A RuntimeError will be raised if modification is attempted. You cannot unfreeze a frozen object. See also Object#frozen?.

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

produces:

```
prog.rb:3:in `<main>': can't modify frozen array (RuntimeError)
```

frozen?

obj.frozen? → true or false

Returns the freeze status of *obj*.

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

hash

obj.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 by class Hash. Any hash value that exceeds the capacity of a Fixnum will be truncated before being used. For instances of class Object, the hash is also the object_id. This will not always be the case for subclasses.

__id__

obj.__id__ → *fixnum*

1.9

Synonym for Object#object_id.

initialize_copy

obj.initialize_copy(*other*) → *other_obj* or *obj*

Part of the protocol used by Object#dup and Object#clone, initialize_copy is invoked as a callback, which should copy across any state information that dup and clone cannot copy themselves. For example, in the following code, a and b reference two instances of the container class, but each instance shares a single string object:

```
class Container
  attr_accessor :content
end
a = Container.new
a.content = "cat"
b = a.dup
a.content[1..-1] = "anary"
a.content   # =>   "canary"
b.content   # =>   "canary"
```

The next example uses initialize_copy to create a new string in the duplicated object.

```
class Container
  attr_accessor :content
  def initialize_copy(other)
    @content = String.new(other.content)
  end
end
a = Container.new
a.content = "cat"
b = a.dup
a.content[1..-1] = "anary"
a.content   # =>   "canary"
b.content   # =>   "cat"
```

inspect

obj.inspect → *string*

Returns a string containing a human-readable representation of *obj*. For objects classes written in Ruby, displays the values of instance variables along with the class name if any instance variables exist. In other cases, uses the to_s method to generate the string. Often this is overridden in child classes to provide class-specific information.

```
[ 1, 2, 3..4, 'five' ].inspect   # =>   [1, 2, 3..4, "five"]
Time.new.inspect                 # =>   2009-04-13 13:26:31 -0500
class Demo
  def initialize
    @a, @b = 1, 2
  end
end
Demo.new.inspect                 # =>   #<Demo:0x0a33a4 @a=1, @b=2>
```

instance_of?

obj.instance_of?(*klass*) → true or false

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

instance_variable_defined?

obj.instance_variable_defined?(*name*) → true or false

Returns true if the named variable is defined. Note that a common idiom, testing to see whether @fred is nil, is incorrect in two ways: first the variable could be defined but set to nil, and second it will generate a warning if debug mode is enabled.

```
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

obj.instance_variable_get(*symbol*) → *other_obj*

Returns the value of the given instance variable (or throws a NameError exception). The @ part of the variable name should be included for regular instance variables.

```
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

obj.instance_variable_set(*symbol*, *other_obj*) → *other_obj*

Sets the instance variable names by *symbol* to *other_obj*, thereby frustrating the efforts of the class's author to attempt to provide proper encapsulation.

```
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.inspect                             # =>   "#<Fred:0x0a3c64
                                                @a=\"dog\", @b=99>"
```

instance_variables

obj.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?

obj.is_a?(*klass*) → true or false

Synonym for Object#kind_of?.

kind_of?

obj.kind_of?(*klass*) → true or false

Returns true if *klass* is the class of *obj* or if *klass* 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.instance_of? A   # =>   false
b.instance_of? B   # =>   true
b.instance_of? C   # =>   false
b.instance_of? M   # =>   false
b.kind_of? A       # =>   true
b.kind_of? B       # =>   true
b.kind_of? C       # =>   false
b.kind_of? M       # =>   true
```

method

obj.method(*symbol*) → *meth*

Looks up the named method 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"
```

methods

obj.methods(*regular*=true) → *array*

If *regular* is true, returns a list of the names of methods publicly accessible in *obj* and *obj*'s ancestors. Otherwise, returns a list of *obj*'s singleton methods.

```
class Klass
  def my_method()
  end
end
k = Klass.new
def k.single
end
k.methods[0..9]    # =>   [:single, :my_method, :nil?, :===, :=~, :!~,
                          :eql?, :class, :clone, :dup]
k.methods.length   # =>   54
k.methods(false)   # =>   [:single]
```

nil?

obj.nil? → true or false

All objects except nil return false.

object_id

obj.object_id → *fixnum*

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. Object#object_id is a different concept from the :name notation, which returns the symbol ID of name. Replaces the deprecated Object#id.

private_methods

obj.private_methods → *array*

Returns a list of private methods accessible within *obj*. This will include the private methods in *obj*'s ancestors, along with any mixed-in module functions.

protected_methods

obj.protected_methods → *array*

Returns the list of protected methods accessible to *obj*.

public_method

obj.public_method(*symbol*) → *meth*

1.9 Looks up the named public method in *obj*, returning a Method object (or raising NameError if the method if not found or if it is found but not public).

```
class Demo
  def initialize(n)
    @iv = n
  end
  def hello()
    puts "Hello, @iv = #{@iv}"
  end
end
k = Demo.new(99)
m = k.public_method(:hello)
m.call
l = Demo.new('Fred')
m = l.public_method(:initialize)
m.call
```

produces:

```
Hello, @iv = 99
prog.rb:15:in `public_method': undefined private method `initialize' for class
`Demo' (NameError)
from /tmp/prog.rb:15:in `<main>'
```

public_methods

obj.public_methods → *array*

Synonym for Object#methods.

public_send

obj.public_send(*name*, ⟨ args ⟩$^+$) → *obj*

1.9 Invokes *obj*'s public method *name*, passing in any arguments. Returns the value returned by the method. See also send, which will also call private and protected methods.

respond_to?

obj.respond_to?(*symbol*, *include_priv*=false) → true or false

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

send

obj.send(*symbol* ⟨ , *args* ⟩* ⟨ , *&block* ⟩) → *other_obj*

Invokes the method identified by *symbol*, passing it any arguments and block. You can use BasicObject#__send__ if the name send clashes with an existing method in *obj*.

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

singleton_methods

obj.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*. (The parameter defaults to false in versions of Ruby prior to January 2004.)

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

taint

obj.taint → *obj*

Marks *obj* as tainted. If the $SAFE level is greater than zero, some objects will be tainted on creation. See Chapter 26, which begins on page 436.

tainted?

obj.tainted? → true or false

Returns true if the object is tainted.

```
a = "cat"
a.tainted?   # =>   false
a.taint      # =>   "cat"
a.tainted?   # =>   true
a.untaint    # =>   "cat"
a.tainted?   # =>   false
```

tap

obj.tap {| *val* | *block* } → *obj*

1.9

Invokes the block, passing *obj* as a parameter. Returns *obj*. Allows you to write code that takes part in a method chain but that does not affect the overall value of the chain.

```
puts "dog"
   .reverse
       .tap {|o| puts "Reversed: #{o}"}
   .capitalize
```

produces:

```
Reversed: god
God
```

to_enum

obj.to_enum(*using*=:each, ⟨ args ⟩$^+$ → *enumerator*

1.9

Returns an Enumerator object that will traverse the content of *obj*. By default, this enumerator will invoke the each method of self, but this can be overridden by passing a different method name as the first parameter. Any additional arguments passed to to_enum will be passed to the enumerator method.

```
by_bytes = "cat".to_enum(:each_byte)
by_bytes.next   # =>   99
by_bytes.next   # =>   97
by_chars = "cat".to_enum(:each_char)
by_chars.next   # =>   "c"
by_chars.next   # =>   "a"
```

to_s

obj.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."

trust

obj.trust → *obj*

1.9

Marks *obj* as trusted. (See the section on trust starting on page 438.)

untaint

obj.untaint → *obj*

Removes the taint from *obj*.

untrust

obj.untrust → *obj*

1.9 Marks *obj* as untrusted. (See the section on trust starting on page 438.)

untrusted?

obj.untrusted → true or false

1.9 Returns true is *obj* is untrusted, false otherwise.

Private instance methods

initialize

initialize(⟨ *arg* ⟩$^+$)

Called as the third and final step in object construction, initialize is responsible for setting up the initial state of the new object. You use the initialize method the same way you'd use constructors in other languages. If you subclass classes other than Object, you will probably want to call super to invoke the parent's initializer.

```
class A
   def initialize(p1)
     puts "Initializing A: p1 = #{p1}"
     @var1 = p1
   end
end
class B < A
  attr_reader :var1, :var2
  def initialize(p1, p2)
    super(p1)
    puts "Initializing B: p2 = #{p2}"
    @var2 = p2
  end
end
b = B.new("cat", "dog")
puts b.inspect
```

produces:

```
Initializing A: p1 = cat
Initializing B: p2 = dog
#<B:0x0a2ea4 @var1="cat", @var2="dog">
```

remove_instance_variable

remove_instance_variable(*symbol*) → *other_obj*

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

```
class Dummy
  def initialize
    @var = 99
  end
  def remove
    remove_instance_variable(:@var)
  end
  def var_defined?
    defined? @var
  end
end
d = Dummy.new
d.var_defined?   # =>  "instance-variable"
d.remove         # =>  99
d.var_defined?   # =>  nil
```

singleton_method_added

singleton_method_added(*symbol*)

Invoked as a callback whenever a singleton method is added to the receiver.

```
module Chatty
  def Chatty.singleton_method_added(id)
    puts "Adding #{id.id2name} to #{self.name}"
  end
  def self.one()     end
  def two()          end
end
def Chatty.three() end
obj = "cat"
def obj.singleton_method_added(id)
  puts "Adding #{id.id2name} to #{self}"
end
def obj.speak
  puts "meow"
end
```

produces:

```
Adding singleton_method_added to Chatty
Adding one to Chatty
Adding three to Chatty
Adding singleton_method_added to cat
Adding speak to cat
```

singleton_method_removed

singleton_method_removed(*symbol*)

Invoked as a callback whenever a singleton method is removed from the receiver.

```
module Chatty
  def Chatty.singleton_method_removed(id)
    puts "Removing #{id.id2name}"
  end
  def self.one()     end
  def two()          end
  def Chatty.three() end
  class <<self
    remove_method :three
    remove_method :one
  end
end
```

produces:

```
Removing three
Removing one
```

singleton_method_undefined

singleton_method_undefined(*symbol*)

Invoked as a callback whenever a singleton method is undefined in the receiver.

```
module Chatty
  def Chatty.singleton_method_undefined(id)
    puts "Undefining #{id.id2name}"
  end
  def Chatty.one()   end
  class << self
     undef_method(:one)
  end
end
```

produces:

```
Undefining one
```