Class Module < Object

Subclasses: Class

A Module is a collection of methods and constants. The methods in a module may be instance methods or module methods. Instance methods appear as methods in a class when the module is included; module methods do not. Conversely, module methods may be called without creating an encapsulating object, and instance methods may not. See also Module#module_function on page 609.

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

```
module Mod
  include Math
  CONST = 1
  def meth
    #  ...
  end
end
Mod.class              # =>  Module
Mod.constants          # =>  [:CONST, :PI, :E]
Mod.instance_methods   # =>  [:meth]
```


Class methods

constants

Module.constants → *array*
Module.constants(*include_parents*) → *array*

1.9

With no argument returns a list of the top-level constants in the interpreter. With one argument, returns the constants defined in class Module (and its parents if the argument is true). This somewhat obscure interface is because Module is a kind of Class, and Class is a subclass of Module. The first form of call is a true call to the class method constants, while the second form actually proxies to the instance method form (see Module#constants later in this section).

```
module Mixin
  CONST_MIXIN = 1
end
class Module
  include Mixin
  SPURIOUS_CONSTANT = 2
end
Module.constants.sort[1..5]              # =>  [:ARGV, :ArgumentError,
                                             :Array, :BasicObject,
                                             :Bignum]
Module.constants.include? :CONST_MIXIN   # =>  false
Module.constants(false)                  # =>  [:SPURIOUS_CONSTANT]
Module.constants(true)                   # =>  [:SPURIOUS_CONSTANT,
                                             :CONST_MIXIN]
```

nesting Module.nesting → *array*

Returns the list of Modules nested at the point of call.

```
module M1
  module M2
    nest = Module.nesting
    p nest
    p nest[0].name
  end
end
```

produces:

```
[M1::M2, M1]
"M1::M2"
```

new Module.new → *mod*
Module.new {| *mod* | *block* } → *mod*

Creates a new anonymous module. If a block is given, it is passed the module object, and the block is evaluated in the context of this module using module_eval.

```
Fred = Module.new do
  def meth1
    "hello"
  end
  def meth2
    "bye"
  end
end
a = "my string"
a.extend(Fred)   # =>   "my string"
a.meth1          # =>   "hello"
a.meth2          # =>   "bye"
```

Instance methods

<, <=, >, >= *mod relop module* → true or false

Hierarchy Query—One module is considered *greater than* another if it is included in (or is a parent class of) the other module. The other operators are defined accordingly. If there is no relationship between the modules, all operators return false.

```
module Mixin
end
module Parent
  include Mixin
end
module Unrelated
end
```

```
Parent > Mixin       # =>   false
Parent < Mixin       # =>   true
Parent <= Parent     # =>   true
Parent < Unrelated   # =>   nil
Parent > Unrelated   # =>   nil
```

<=> *mod* <=> *other_mod* → −1, 0, +1

Comparison—Returns −1 if *mod* includes *other_mod*, 0 if *mod* is the same module as *other_mod*, and +1 if *mod* is included by *other_mod* or if *mod* has no relationship with *other_mod*.

=== *mod* === *obj* → true or false

Case Equality—Returns true if *obj* is an instance of *mod* or one of *mod*'s descendents. Of limited use for modules but can be used in case statements to test objects by class.

ancestors *mod*.ancestors → *array*

Returns a list of modules included in *mod* (including *mod* itself).

```
module Mod
  include Math
  include Comparable
end

Mod.ancestors    # =>   [Mod, Comparable, Math]
Math.ancestors   # =>   [Math]
```

autoload *mod*.autoload(*name*, *file_name*) → nil

Registers *file_name* to be loaded (using Kernel.require) the first time that module *name* (which may be a String or a Symbol) is accessed in the namespace of *mod*. Note that the autoloaded file is evaluated in the top-level context. In this example, module_b.rb contains the following:

```
module A::B    # in module_b.rb
  def doit
    puts "In Module A::B"
  end
  module_function :doit
end
```

Other code can then include this module automatically.

```
module A
  autoload(:B, "module_b")
end
A::B.doit            # autoloads "module_b"
```

produces:

```
In Module A::B
```

autoload? *mod*.autoload?(*name*) → *file_name* or nil

Returns the name of the file that will be autoloaded when the string or symbol *name* is referenced in the context of *mod*or returns nil if there is no associated autoload.

```
module A
  autoload(:B, "module_b")
end
A.autoload?(:B)   # =>   "module_b"
A.autoload?(:C)   # =>   nil
```

class_eval *mod*.class_eval(*string* ⟨ , *file_name* ⟨ , *line_number* ⟩ ⟩) → *obj*
mod.class_eval { *block* } → *obj*

Synonym for Module.module_eval.

class_exec *mod*.class_exec(⟨ args ⟩+) {| *args* | *block* } → *obj*

1.9 Synonym for Module.module_exec.

class_variable_defined? *mod*.class_variable_defined?(*name*) → true or false

1.9 Returns true if the named class variable is defined in *mod*. The two @ signs are a required part of the name.

```
class One
  @@var1 = "wibble"
end
One.class_variable_defined?(:@@var1)   # =>   true
One.class_variable_defined?(:@@var2)   # =>   false
```

class_variable_get *mod*.class_variable_get(*name*) → *obj*

1.9 Returns the value of the named class variable. The two @ signs must appear in the name.

```
class One
  @@var1 = "wibble"
end
One.class_variable_get(:@@var1)    # =>   "wibble"
One.class_variable_get("@@var1")   # =>   "wibble"
```

class_variable_set *mod*.class_variable_set(*name*, *value*) → *value*

1.9 Returns the value of the named class variable. The two @ signs must appear in the name.

```
class One
  @@var1 = "wibble"
end
One.class_variable_set(:@@var1, 99)   # =>   99
One.class_variable_get("@@var1")      # =>   99
```

class_variables *mod*.class_variables → *array*

1.9 Returns an array of the names of class variables in *mod*. (As of Ruby 1.9 class variables are

no longer shared with child classes, so this listing is restricted to the class variables defined in *mod*.)

```
class One
  @@var1 = 1
end
class Two < One
  @@var2 = 2
end
One.class_variables   # =>   [:@@var1]
Two.class_variables   # =>   [:@@var2]
```

const_defined?

mod.const_defined?(*symbol* ⟨ *search_parents*=true ⟩) → true or false

Returns true if a constant with the given name is defined by *mod* or the parents of *mod* (if the second parameter is true).

```
Math.const_defined? "PI"   # =>   true
```

const_get

mod.const_get(*symbol*) → *obj*

Returns the value of the named constant in *mod*.

```
Math.const_get :PI   # =>   3.14159265358979
```

const_missing

const_missing(*symbol*) → *obj*

Invoked when a reference is made to an undefined constant in *mod*. It is passed a symbol for the undefined constant and returns a value to be used for that constant. The following code is very poor style. If a reference is made to an undefined constant, it attempts to load a file whose name is the lowercase version of the constant (thus, class Fred is assumed to be in file fred.rb). If found, it returns the value of the loaded class. It therefore implements a perverse kind of autoload facility.

```
def Object.const_missing(name)
  @looked_for ||= {}
  str_name = name.to_s
  raise "Class not found: #{name}" if @looked_for[str_name]
  @looked_for[str_name] = 1
  file = str_name.downcase
  require file
  klass = const_get(name)
  return klass if klass
  raise "Class not found: #{name}"
end
```

const_set

mod.const_set(*symbol*, *obj*) → *obj*

Sets the named constant to the given object, returning that object. Creates a new constant if no constant with the given name previously existed.

```
Math.const_set("HIGH_SCHOOL_PI", 22.0/7.0)   # =>   3.14285714285714
Math::HIGH_SCHOOL_PI - Math::PI              # =>   0.00126448926734968
```

constants

mod.constants(*include_parents* = true) → *array*

1.9

Returns an array of the names of the constants accessible in *mod*. If the parameter is true, this includes the names of constants in any included modules.

```
IO.constants(false)       # =>   [:SEEK_SET, :SEEK_CUR, :SEEK_END]
# Now include stuff defined in module File::Constants
IO.constants(true)[1,6]   # =>   [:SEEK_CUR, :SEEK_END, :LOCK_SH,
                                  :LOCK_EX, :LOCK_UN, :LOCK_NB]
```

include?

mod.include?(*other_mod*) → true or false

Returns true if *other_mod* is included in *mod* or one of *mod*'s ancestors.

```
module A
end

class B
  include A
end

class C < B
end

B.include?(A)   # =>   true
C.include?(A)   # =>   true
A.include?(A)   # =>   false
```

included_modules

mod.included_modules → *array*

Returns the list of modules included in *mod*.

```
module Mixin
end

module Outer
  include Mixin
end

Mixin.included_modules   # =>   []
Outer.included_modules   # =>   [Mixin]
```

instance_method

mod.instance_method(*symbol*) → *unbound_method*

Returns an UnboundMethod representing the given instance method in *mod*.

```
class Interpreter
  def do_a() print "there, "; end
  def do_d() print "Hello ";  end
  def do_e() print "!\n";     end
  def do_v() print "Dave";    end
  Dispatcher = {
    'a' => instance_method(:do_a),
    'd' => instance_method(:do_d),
```

```
    'e' => instance_method(:do_e),
    'v' => instance_method(:do_v)
  }
  def interpret(string)
    string.each_char {|ch| Dispatcher[ch].bind(self).call }
  end
end
interpreter = Interpreter.new
interpreter.interpret('dave')
```

produces:

```
Hello there, Dave!
```

instance_methods

mod.instance_methods(*inc_super*=true) → *array*

Returns an array containing the names of public and protected instance methods in the receiver. For a module, these are the public methods; for a class, they are the instance (not singleton) methods. With no argument or with an argument that is true, the methods in *mod* and *mod*'s superclasses are returned. When called with a module as a receiver or with a parameter that is false, the instance methods in *mod* are returned. (The parameter defaults to false in versions of Ruby prior to January 2004.)

```
module A
  def method1()
  end
end

class B
  def method2()
  end
end

class C < B
  def method3()
  end
end

A.instance_methods                # =>   [:method1]
B.instance_methods(false)         # =>   [:method2]
C.instance_methods(false)         # =>   [:method3]
C.instance_methods(true).length   # =>   54
```

method_defined?

mod.method_defined?(*symbol*) → true or false

Returns true if the named method is defined by *mod* (or its included modules and, if *mod* is a class, its ancestors). Public and protected methods are matched.

```
module A
  def method1()  end
end
class B
  def method2()  end
end
class C < B
  include A
  def method3()  end
end

A.method_defined? :method1    # =>  true
C.method_defined? "method1"   # =>  true
C.method_defined? "method2"   # =>  true
C.method_defined? "method3"   # =>  true
C.method_defined? "method4"   # =>  false
```

module_eval

mod.class_eval(*string* ⟨ , *file_name* ⟨ , *line_number* ⟩ ⟩) → *obj*
mod.module_eval { *block* } → *obj*

Evaluates the string or block in the context of *mod*. This can be used to add methods to a class. module_eval returns the result of evaluating its argument. The optional *file_name* and *line_number* parameters set the text for error messages.

```
class Thing
end
a = %q{def hello() "Hello there!" end}
Thing.module_eval(a)
puts Thing.new.hello()
Thing.module_eval("invalid code", "dummy", 123)
```

produces:

```
Hello there!
dummy:123:in `<main>': undefined local variable
    or method `code' for Thing:Class
```

module_exec

mod.module_exec(⟨ args ⟩+) {| *args* | *block* } → *obj*

1.9

Behaves the same as the block form for Module#module_eval, except any parameters passed to the method are in turn passed to the block. This gives you a way of passing in values that would otherwise not be in scope in the block (because *self* is changed).

```
class Thing
end
name  = :new_instance_variable
Thing.module_exec(name) do |iv_name|
  attr_accessor iv_name
end
t = Thing.new
t.new_instance_variable = "wibble"
p t
```

produces:

```
#<Thing:0x0a4268 @new_instance_variable="wibble">
```

name *mod*.name → *string*

Returns the name of the module *mod*.

private_class_method *mod*.private_class_method(⟨ *symbol* ⟩+) → nil

Makes existing class methods private. Often used to hide the default constructor new.

```
class SimpleSingleton  # Not thread safe
  private_class_method :new
  def SimpleSingleton.create(*args, &block)
    @me = new(*args, &block) if ! @me
    @me
  end
end
```

private_instance_methods

mod.private_instance_methods(*inc_super*=true) → *array*

Returns a list of the private instance methods defined in *mod*. If the optional parameter is true, the methods of any ancestors are included. (The parameter defaults to false in versions of Ruby prior to January 2004.)

```
module Mod
  def method1()  end
  private :method1
  def method2()  end
end
Mod.instance_methods           # =>   [:method2]
Mod.private_instance_methods   # =>   [:method1]
```

private_method_defined? *mod*.private_method_defined?(*symbol*) → true or false

Returns true if the named private method is defined by *mod* (or its included modules and, if *mod* is a class, its ancestors).

```
module A
  def method1()  end
end
class B
  private
  def method2()  end
end
class C < B
  include A
  def method3()  end
end

A.method_defined? :method1            # =>   true
C.private_method_defined? "method1"   # =>   false
C.private_method_defined? "method2"   # =>   true
C.method_defined? "method2"           # =>   false
```

protected_instance_methods *mod*.protected_instance_methods(*inc_super*=true) → *array*

Returns a list of the protected instance methods defined in *mod*. If the optional parameter is true, the methods of any ancestors are included. (The parameter defaults to false in versions of Ruby prior to January 2004.)

protected_method_defined? *mod*.protected_method_defined?(*symbol*) → true or false

Returns true if the named protected method is defined by *mod* (or its included modules and, if *mod* is a class, its ancestors).

```
module A
  def method1()  end
end
class B
  protected
  def method2()  end
end
class C < B
  include A
  def method3()  end
end

A.method_defined? :method1              # =>   true
C.protected_method_defined? "method1"   # =>   false
C.protected_method_defined? "method2"   # =>   true
C.method_defined? "method2"             # =>   true
```

public_class_method *mod*.public_class_method(⟨ *symbol* ⟩+) → nil

Makes a list of existing class methods public.

public_instance_method *mod*.public_instance_method(*symbol*) → *unbound_method*

1.9

Returns an UnboundMethod representing the given public instance method in *mod*. See also Module#instance_method, which ignores scope.

```
class Test
  def method_a; end
private
  def method_b; end
end
puts "method_a is #{Test.public_instance_method(:method_a)}"
puts "method_b is #{Test.public_instance_method(:method_b)}"
```

produces:

```
method_a is #<UnboundMethod: Test#method_a>
prog.rb:7:in `public_instance_method': undefined private method `method_b' for
class `Test' (NameError)
from /tmp/prog.rb:7:in `<main>'
```

public_instance_methods

mod.public_instance_methods(*inc_super*=true) → *array*

Returns a list of the public instance methods defined in *mod*. If the optional parameter is true, the methods of any ancestors are included. (The parameter defaults to false in versions of Ruby prior to January 2004.)

public_method_defined?

mod.public_method_defined?(*symbol*) → true or false

Returns true if the named public method is defined by *mod* (or its included modules and, if *mod* is a class, its ancestors).

```
module A
  def method1()  end
end
class B
  protected
  def method2()  end
end
class C < B
  include A
  def method3()  end
end

A.method_defined? :method1           # =>   true
C.public_method_defined? "method1"   # =>   true
C.public_method_defined? "method2"   # =>   false
C.method_defined? "method2"          # =>   true
```

remove_class_variable

remove_class_variable(*symbol*) → *obj*

1.9

Removes the definition of the *symbol*, returning that variable's value. Prior to Ruby 1.9, this method was private.

```
class Dummy
  @@var = 99
end
Dummy.class_eval { p defined? @@var }
puts Dummy.remove_class_variable(:@@var)
Dummy.class_eval { p defined? @@var }
```

produces:

```
"class variable"
99
nil
```

Private instance methods

alias_method

alias_method(*new_id, old_id*) → *mod*

Makes *new_id* a new copy of the method *old_id*. This can be used to retain access to methods that are overridden.

```
module Mod
  alias_method :orig_exit, :exit
  def exit(code=0)
    puts "Exiting with code #{code}"
    orig_exit(code)
  end
end
include Mod
exit(99)
```

produces:

```
Exiting with code 99
```

append_features

append_features(*other_mod*) → *mod*

When this module is included in another, Ruby calls append_features in this module, passing it the receiving module in *other_mod*. Ruby's default implementation is to add the constants, methods, and module variables of this module to *other_mod* if this module has not already been added to *other_mod* or one of its ancestors. Prior to Ruby 1.8, user code often redefined append_features, added its own functionality, and then invoked super to handle the real include. Now you should instead implement the method Module#included, described on page 608.

attr

attr(⟨ symbol ⟩+) → nil

1.9

An alias for Module#attr_reader as of Ruby 1.9.

attr_accessor

attr_accessor(⟨ *symbol* ⟩+) → nil

Creates a reader and a writer method for each symbol passed as an argument. These methods provide access to the underlying instance variables of the name name (with a leading @ sign).

```
class Test
  attr_accessor :name, :likes
  def initialize(name, likes)
    @name = name
    @likes = likes
  end
end
d = Test.new("Dave", "Ruby")
d.name = "Chad"
d.name    # =>   "Chad"
d.likes   # =>   "Ruby"
```

attr_reader

attr_reader(⟨ *symbol* ⟩+) → nil

Creates instance variables and corresponding methods that return the value of each instance variable.

```
class Test
  attr_reader :name, :likes
  def initialize(name, likes)
    @name = name
    @likes = likes
  end
end
d = Test.new("Dave", "Ruby")
d.name    # =>   "Dave"
d.likes   # =>   "Ruby"
```

attr_writer

attr_writer(⟨ *symbol* ⟩+) → nil

Creates an accessor method to allow assignment to the attribute *symbol*.id2name.

```
class Test
  attr_writer :name, :likes
  def initialize(name, likes)
    @name = name
    @likes = likes
  end
end
d = Test.new("Dave", "Ruby")
d.name = "Chad"
d   # =>   #<Test:0x0a3c28 @name="Chad", @likes="Ruby">
```

define_method

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

Defines an instance 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. This is tricky to demonstrate because define_method is private. (This is why we resort to the send hack in this example.) See also Object#define_singleton_method.

```
class A
  def fred
    puts "In Fred"
  end
  def create_method(name, &block)
    self.class.send(:define_method, name, &block)
  end
  define_method(:wilma) { puts "Charge it!" }
end
class B < A
  define_method(:barney, instance_method(:fred))
end
b = B.new
b.barney
b.wilma
```

```
b.create_method(:betty) { p self }
b.betty
```

produces:

```
In Fred
Charge it!
#<B:0x0a2454>
```

1.9 Note that it is possible to define methods with names that are not valid if you were to use the def keyword. This methods can not be invoked directly.

```
class Silly
  define_method("Oh !@!#^!") { puts "As Snoopy says" }
end
Silly.new.send("Oh !@!#^!")
```

produces:

```
As Snoopy says
```

extend_object

extend_object(*obj*) → *obj*

Extends the specified object by adding this module's constants and methods (which are added as singleton methods). This is the callback method used by Object#extend.

```
module Picky
  def Picky.extend_object(o)
    if String === o
      puts "Can't add Picky to a String"
    else
      puts "Picky added to #{o.class}"
      super
    end
  end
end
(s = Array.new).extend Picky  # Call Object.extend
(s = "quick brown fox").extend Picky
```

produces:

```
Picky added to Array
Can't add Picky to a String
```

extended

extended(*other_mod*)

Callback invoked whenever the receiver is used to extend an object. The object is passed as a parameter. This should be used in preference to Module#extend_object if your code wants to perform some action when a module is used to extend an object.

```
module A
  def A.extended(obj)
    puts "#{self} extending '#{obj}'"
  end
end
"cat".extend(A)
```

produces:

```
A extending 'cat'
```

include include(⟨ *other_mod* ⟩+) → *mod*

Invokes Module.append_features (documented on page 605) on each parameter (in reverse order). Equivalent to the following code:

```
def include(*modules)
  modules.reverse_each do |mod|
    mod.append_features(self)
    mod.included(self)
  end
end
```

included included(*other_mod*)

Callback invoked whenever the receiver is included in another module or class. This should be used in preference to Module#append_features if your code wants to perform some action when a module is included in another.

```
module A
  def A.included(mod)
    puts "#{self} included in #{mod}"
  end
end
module Enumerable
  include A
end
```

produces:

```
A included in Enumerable
```

method_added method_added(*symbol*)

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

```
module Chatty
  def Chatty.method_added(id)
    puts "Adding #{id.id2name}"
  end
  def one()   end
end
module Chatty
  def two()   end
end
```

produces:

```
Adding one
Adding two
```

method_removed method_removed(*symbol*)

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

```
module Chatty
  def Chatty.method_removed(id)
    puts "Removing #{id.id2name}"
```

```
  end
  def one()   end
end
module Chatty
  remove_method(:one)
end
```

produces:

```
Removing one
```

method_undefined

method_undefined(*symbol*)

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

```
module Chatty
  def Chatty.method_undefined(id)
    puts "Undefining #{id.id2name}"
  end
  def one()   end
end
module Chatty
  undef_method(:one)
end
```

produces:

```
Undefining one
```

module_function

module_function(⟨ *symbol* ⟩*) → *mod*

Creates module functions for the named methods. These functions may be called with the module as a receiver and are available as instance methods to classes that mix in the module. Module functions are copies of the original and so may be changed independently. The instance-method versions are made private. If used with no arguments, subsequently defined methods become module functions.

```
module Mod
  def one
    "This is one"
  end
  module_function :one
end
class Cls
  include Mod
  def call_one
    one
  end
end
```

```
Mod.one      # =>   "This is one"
c = Cls.new
c.call_one   # =>   "This is one"
module Mod
  def one
    "This is the new one"
  end
end
Mod.one      # =>   "This is one"
c.call_one   # =>   "This is the new one"
```

private private(⟨ *symbol* ⟩*) → *mod*

With no arguments, sets the default visibility for subsequently defined methods to private. With arguments, sets the named methods to have private visibility. See “Access Control” starting on page 362.

```
module Mod
  def a()  end
  def b()  end
  private
  def c()  end
  private :a
end
Mod.private_instance_methods   # =>   [:a, :c]
```

protected protected(⟨ *symbol* ⟩*) → *mod*

With no arguments, sets the default visibility for subsequently defined methods to protected. With arguments, sets the named methods to have protected visibility. See “Access Control” starting on page 362.

public public(⟨ *symbol* ⟩*) → *mod*

With no arguments, sets the default visibility for subsequently defined methods to public. With arguments, sets the named methods to have public visibility. See “Access Control” starting on page 362.

remove_const remove_const(*symbol*) → *obj*

Removes the definition of the given constant, returning that constant’s value. Predefined classes and singleton objects (such as *true*) cannot be removed.

remove_method remove_method(*symbol*) → *mod*

Removes the method identified by *symbol* from the current class. For an example, see Module.undef_method.

undef_method undef_method(⟨ *symbol* ⟩+) → *mod*

Prevents the current class from responding to calls to the named method(s). Contrast this with remove_method, which deletes the method from the particular class; Ruby will still search superclasses and mixed-in modules for a possible receiver.

```
class Parent
  def hello
    puts "In parent"
  end
end
class Child < Parent
  def hello
    puts "In child"
  end
end
c = Child.new
c.hello
class Child
  remove_method :hello  # remove from child, still in parent
end
c.hello
class Child
  undef_method :hello   # prevent any calls to 'hello'
end
c.hello
```

produces:

```
In child
In parent
prog.rb:23:in `<main>': undefined method `hello' for #<Child:0x0a3048>
(NoMethodError)
```