Home

Pages Classes Methods

Search

Show/hide navigation

Parent

Methods

::new
#!
#!=
#==
#__id__
#__send__
#equal?
#instance_eval
#instance_exec
#method_missing
#singleton_method_added
#singleton_method_removed
#singleton_method_undefined

class BasicObject

`BasicObject` is the parent class of all classes in Ruby. It's an explicit blank class.

`BasicObject` can be used for creating object hierarchies independent of Ruby's object hierarchy, proxy objects like the Delegator class, or other uses where namespace pollution from Ruby's methods and classes must be avoided.

To avoid polluting BasicObject for other users an appropriately named subclass of BasicObject should be created instead of directly modifying BasicObject:

```
class MyObjectSystem < BasicObject
end
```

BasicObject does not include Kernel (for methods like puts) and BasicObject is outside of the namespace of the standard library so common classes will not be found without using a full class path.

A variety of strategies can be used to provide useful portions of the standard library to subclasses of BasicObject. A subclass could include Kernel to obtain puts, exit, etc. A custom Kernel-like module could be created and included or delegation can be used via method_missing:

```
class MyObjectSystem < BasicObject
  DELEGATE = [:puts, :p]

  def method_missing(name, *args, &block)
    return super unless DELEGATE.include? name
    ::Kernel.send(name, *args, &block)
  end

  def respond_to_missing?(name, include_private = false)
    DELEGATE.include?(name) or super
  end
end
```

Access to classes and modules from the Ruby standard library can be obtained in a BasicObject subclass by referencing the desired constant from the root like ::File or ::Enumerator. Like method_missing, const_missing can be used to delegate constant lookup to Object:

```
class MyObjectSystem < BasicObject
  def self.const_missing(name)
    ::Object.const_get(name)
  end
end
```

What's Here

These are the methods defined for BasicObject:

- ::new: Returns a new BasicObject instance.
- #!: Returns the boolean negation of self: true or false.
- #!=: Returns whether self and the given object are *not* equal.
- ==: Returns whether self and the given object are equivalent.
- #__id__: Returns the integer object identifier for self.

- #__send__: Calls the method identified by the given symbol.
- `equal?` : Returns whether `self` and the given object are the same object.
- `instance_eval` : Evaluates the given string or block in the context of `self` .
- `instance_exec` : Executes the given block in the context of `self` , passing the given arguments.

Public Class Methods

new

Returns a new `BasicObject` .

Public Instance Methods

!obj → true or false

Boolean negate.

obj != other → true or false

Returns true if two objects are not-equal, otherwise false.

obj == 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
```

Also aliased as: equal?

`__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
```

`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"
```

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
```

Alias for: `==`

instance_eval(string [, filename [, lineno]]) → obj
instance_eval {|obj| block } → obj

Evaluates a string containing Ruby source code, or the given block, within the context of the receiver (*obj*). In order to set the context, the variable `self` is set to *obj* while the code is executing, giving the code access to *obj*'s instance variables and private methods.

When `instance_eval` is given a block, *obj* is also passed in as the block's only argument.

When `instance_eval` is given a `String`, the optional second and third parameters supply a filename and starting line number that are used when reporting compilation errors.

```
class KlassWithSecret
  def initialize
    @secret = 99
  end
  private
  def the_secret
    "Ssssh! The secret is #{@secret}."
  end
end
k = KlassWithSecret.new
k.instance_eval { @secret }          #=> 99
k.instance_eval { the_secret }       #=> "Ssssh! The secret is 99."
k.instance_eval {|obj| obj == self } #=> true
```

`instance_exec(arg...) {|var...| block } → obj`

Executes the given block within the context of the receiver (*obj*). In order to set the context, the variable `self` is set to *obj* while the code is executing, giving the code access to *obj*'s instance variables. Arguments are passed as block parameters.

```
class KlassWithSecret
  def initialize
    @secret = 99
  end
end
k = KlassWithSecret.new
k.instance_exec(5) {|x| @secret+x }   #=> 104
```

Private Instance Methods

`method_missing(symbol [, *args] ) → result`

Invoked by Ruby when *obj* is sent a message it cannot handle. *symbol* is the symbol for the method called, and *args* are any arguments that were passed to it. By default, the interpreter raises an error when this method is called. However, it is possible to override the method to provide more dynamic behavior. If it is decided that a particular method should not be handled, then *super* should be called, so that ancestors can pick up the missing method. The example below creates a class `Roman`, which responds to methods with names consisting of roman numerals, returning the corresponding integer values.

```
class Roman
  def roman_to_int(str)
    # ...
```

```
  end

  def method_missing(symbol, *args)
    str = symbol.id2name
    begin
      roman_to_int(str)
    rescue
      super(symbol, *args)
    end
  end
end

r = Roman.new
r.iv      #=> 4
r.xxiii   #=> 23
r.mm      #=> 2000
r.foo     #=> NoMethodError
```

`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}"
  end
  def self.one()     end
  def two()          end
  def Chatty.three() end
end
```

produces:

```
Adding singleton_method_added
Adding one
Adding three
```

`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(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
```