Class

BasicObject

1.9

BasicObject is the root of Ruby's class hierarchy. It deliberately has just a few methods, allowing it to be conveniently used as the basis for a number of metaprogramming techniques.

If you write code in a direct descendent of BasicObject, you will not have unqualified access to the methods in Kernel, which normally get mixed in to Object. This example illustrates how to invoke Kernel methods explicitly:

```
class SimpleBuilder < BasicObject
  def __puts_at_indent__(string)
    ::Kernel.puts " " * @indent + string
  end
  def method_missing(name, *args, &block)
    @indent ||= 0
    __puts_at_indent__("<#{name}>")
    @indent += 2
    __puts_at_indent__(args.join) unless args.empty?
    yield if ::Kernel.block_given?
    @indent -= 2
    __puts_at_indent__("</#{name}>")
  end
end
r = SimpleBuilder.new
r.person do
  r.name "Dave"
  r.address do
    r.street "123 Main"
    r.city   "Pleasantville"
  end
end
```

produces:

```
<person>
  <name>
    Dave
  </name>
  <address>
    <street>
      123 Main
    </street>
    <city>
      Pleasantville
    </city>
  </address>
</person>
```

Instance methods

! ! *obj* → true or false

Returns false unless *obj* is false. It is defined in BasicObject so ! is defined for all objects in Ruby.

== *obj* == *other_obj* → true or false

Equality—At the BasicObject level, == returns true only if *obj* and *other_obj* are the same object. Typically, this method is overridden in descendent classes to provide class-specific meaning.

!= *obj* != *other* → true or false

Returns the opposite of BasicObject#==.

equal? *obj*.equal?(*other_obj*) → true or false

Alias for BasicObject#==.

instance_eval *obj*.instance_eval(*string* ⟨ , *file* ⟨ , *line* ⟩ ⟩) → *other_obj*
obj.instance_eval { *block* } → *other_obj*

Evaluates a string containing Ruby source code, or the given block, within the context of the receiver (*obj*). 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. In the version of instance_eval that takes a String, the optional second and third parameters supply a filename and starting line number that are used when reporting compilation errors.

```
class Klass
  def initialize
    @secret = 99
  end
end
k = Klass.new
k.instance_eval { @secret }   # =>   99
```

When metaprogramming, instance_eval is often used to execute the methods in a block in the context of the caller:

```
class Recorder < BasicObject
  attr_reader :__calls__
  def method_missing(name, *args, &block)
    @__calls__ ||= []
    @__calls__ << [ name, args ]
  end
  def record(&block)
    instance_eval(&block)
  end
end
```

```
r = Recorder.new
r.record do
  disable "safety"
  pull    "control rod", dir: "out"
  run
end
p r.__calls__
```

produces:

```
[[:disable, ["safety"]], [:pull, ["control rod", {:dir=>"out"}]], [:run, []]]
```

instance_exec

obj.instance_exec(⟨ args ⟩* {|*args*| *block* } → *other_obj*)

1.9

Executes the block with self set to *obj*, passing *args* as parameters to the block.

```
class Dummy < BasicObject
  def initialize
    @iv = 33
  end
  def double_and_call(value, &block)
    instance_exec(value*2, &block)
  end
end
d = Dummy.new
d.double_and_call(22) do |param|
  ::Kernel::puts "Parameter = #{param}"
  ::Kernel::puts "@iv = #{@iv}"
end
```

produces:

```
Parameter = 44
@iv = 33
```

method_missing

obj.method_missing(*symbol* ⟨ , **args* ⟩) → *other_obj*

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. method_missing can be used to implement proxies, delegators, and forwarders. It can also be used to simulate the existence of methods in the receiver, as the example at the start of this section shows.

__send__

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

Invokes the method identified by *symbol*, passing it any arguments and block.

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