Home

Pages Classes Methods

Search

Table of Contents

What's Here
Converting
Querying
Exiting
Exceptions
IO
Procs
Tracing
Subprocesses
Loading
Yielding
Random Values
Other

Show/hide navigation

Methods

#Array
#Complex
#Float
#Hash
#Integer
#Rational
#String
#__callee__
#__dir__
#__method__
#`
#abort
#at_exit
#autoload
#autoload?
#binding
#block_given?
#callcc
#caller
#caller_locations
#catch
#chomp
#chop
#class
#clone
#eval
#exec
#exit
#exit!
#fail
#fork
#format
#frozen?

module Kernel

The `Kernel` module is included by class `Object`, so its methods are available in every Ruby object.

The `Kernel` instance methods are documented in class `Object` while the module methods are documented here. These methods are called without a receiver and thus can be called in functional form:

```
sprintf "%.1f", 1.234 #=> "1.2"
```

What's Here

Module Kernel provides methods that are useful for:

- Converting
- Querying
- Exiting
- Exceptions
- IO
- Procs
- Tracing
- Subprocesses
- Loading
- Yielding
- Random Values
- Other

Converting

- `Array`: Returns an `Array` based on the given argument.
- `Complex`: Returns a `Complex` based on the given arguments.
- `Float`: Returns a `Float` based on the given arguments.
- `Hash`: Returns a `Hash` based on the given argument.
- `Integer`: Returns an `Integer` based on the given arguments.
- `Rational`: Returns a `Rational` based on the given arguments.
- `String`: Returns a `String` based on the given argument.

Querying

- #__callee__: Returns the called name of the current method as a symbol.
- #__dir__: Returns the path to the directory from which the current method is called.
- #__method__: Returns the name of the current method as a symbol.
- `autoload?` : Returns the file to be loaded when the given module is referenced.
- `binding` : Returns a `Binding` for the context at the point of call.
- `block_given?` : Returns `true` if a block was passed to the calling method.
- `caller` : Returns the current execution stack as an array of strings.
- `caller_locations` : Returns the current execution stack as an array of `Thread::Backtrace::Location` objects.
- `class` : Returns the class of `self` .
- `frozen?` : Returns whether `self` is frozen.
- `global_variables` : Returns an array of global variables as symbols.
- `local_variables` : Returns an array of local variables as symbols.
- `test` : Performs specified tests on the given single file or pair of files.

Exiting

- `abort` : Exits the current process after printing the given arguments.
- `at_exit` : Executes the given block when the process exits.
- `exit` : Exits the current process after calling any registered `at_exit` handlers.
- `exit!` : Exits the current process without calling any registered `at_exit` handlers.

Exceptions

- `catch` : Executes the given block, possibly catching a thrown object.
- `raise` (aliased as `fail`): Raises an exception based on the given arguments.
- `throw` : Returns from the active catch block waiting for the given tag.

IO

- ::pp: Prints the given objects in pretty form.
- `gets` : Returns and assigns to `$_` the next line from the current input.

- `open` : Creates an `IO` object connected to the given stream, file, or subprocess.
- `p` : Prints the given objects' inspect output to the standard output.
- `print` : Prints the given objects to standard output without a newline.
- `printf` : Prints the string resulting from applying the given format string to any additional arguments.
- `putc` : Equivalent to <tt.$stdout.putc(object)</tt> for the given object.
- `puts` : Equivalent to `$stdout.puts(*objects)` for the given objects.
- `readline` : Similar to `gets` , but raises an exception at the end of file.
- `readlines` : Returns an array of the remaining lines from the current input.
- `select` : Same as `IO.select` .

Procs

- `lambda` : Returns a lambda proc for the given block.
- `proc` : Returns a new `Proc` ; equivalent to `Proc.new` .

Tracing

- `set_trace_func` : Sets the given proc as the handler for tracing, or disables tracing if given `nil` .
- `trace_var` : Starts tracing assignments to the given global variable.
- `untrace_var` : Disables tracing of assignments to the given global variable.

Subprocesses

- `command`: Returns the standard output of running `command` in a subshell.
- `exec` : Replaces current process with a new process.
- `fork` : Forks the current process into two processes.
- `spawn` : Executes the given command and returns its pid without waiting for completion.
- `system` : Executes the given command in a subshell.

Loading

- `autoload` : Registers the given file to be loaded when the given constant is first referenced.
- `load` : Loads the given Ruby file.
- `require` : Loads the given Ruby file unless it has already been loaded.

- `require_relative` : Loads the Ruby file path relative to the calling file, unless it has already been loaded.

Yielding

- `tap` : Yields `self` to the given block; returns `self` .
- `then` (aliased as `yield_self`): Yields `self` to the block and returns the result of the block.

Random Values

- `rand` : Returns a pseudo-random floating point number strictly between 0.0 and 1.0.
- `srand` : Seeds the pseudo-random number generator with the given number.

Other

- `eval` : Evaluates the given string as Ruby code.
- `loop` : Repeatedly executes the given block.
- `sleep` : Suspends the current thread for the given number of seconds.
- `sprintf` (aliased as `format`): Returns the string resulting from applying the given format string to any additional arguments.
- `syscall` : Runs an operating system call.
- `trap` : Specifies the handling of system signals.
- `warn` : Issue a warning based on the given messages and options.

Public Instance Methods

`Array(object) → object or new_array`

Returns an array converted from `object` .

Tries to convert `object` to an array using `to_ary` first and `to_a` second:

```
Array([0, 1, 2])         # => [0, 1, 2]
Array({foo: 0, bar: 1}) # => [[:foo, 0], [:bar, 1]]
Array(0..4)              # => [0, 1, 2, 3, 4]
```

Returns `object` in an array, `[object]` , if `object` cannot be converted:

```
Array(:foo)              # => [:foo]
```

Complex(x[, y], exception: true) → numeric or nil

Returns x+i*y;

```
Complex(1, 2)    #=> (1+2i)
Complex('1+2i')  #=> (1+2i)
Complex(nil)     #=> TypeError
Complex(1, nil)  #=> TypeError

Complex(1, nil, exception: false)  #=> nil
Complex('1+2', exception: false)   #=> nil
```

Syntax of string form:

```
string form = extra spaces , complex , extra spaces ;
complex = real part | [ sign ] , imaginary part
        | real part , sign , imaginary part
        | rational , "@" , rational ;
real part = rational ;
imaginary part = imaginary unit | unsigned rational , imaginary unit ;
rational = [ sign ] , unsigned rational ;
unsigned rational = numerator | numerator , "/" , denominator ;
numerator = integer part | fractional part | integer part , fractional part ;
denominator = digits ;
integer part = digits ;
fractional part = "." , digits , [ ( "e" | "E" ) , [ sign ] , digits ] ;
imaginary unit = "i" | "I" | "j" | "J" ;
sign = "-" | "+" ;
digits = digit , { digit | "_" , digit };
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
extra spaces = ? \s* ? ;
```

See String#to_c.

Float(arg, exception: true) → float or nil

Returns *arg* converted to a float. Numeric types are converted directly, and with exception to String and `nil` the rest are converted using *arg* `.to_f`. Converting a String with invalid characters will result in a ArgumentError. Converting `nil` generates a TypeError. Exceptions can be suppressed by passing `exception: false`.

```
Float(1)                 #=> 1.0
Float("123.456")         #=> 123.456
Float("123.0_badstring") #=> ArgumentError: invalid value for Float(): "123.0
Float(nil)               #=> TypeError: can't convert nil into Float
Float("123.0_badstring", exception: false)  #=> nil
```

`Hash(object) → object or new_hash`

Returns a hash converted from `object`.

- If `object` is:
  - A hash, returns `object`.
  - An empty array or `nil`, returns an empty hash.
- Otherwise, if `object.to_hash` returns a hash, returns that hash.
- Otherwise, returns `TypeError`.

Examples:

```
Hash({foo: 0, bar: 1}) # => {:foo=>0, :bar=>1}
Hash(nil)              # => {}
Hash([])               # => {}
```

`Integer(object, base = 0, exception: true) → integer or nil`

Returns an integer converted from `object`.

Tries to convert `object` to an integer using `to_int` first and `to_i` second; see below for exceptions.

With a non-zero `base`, `object` must be a string or convertible to a string.

numeric objects

With integer argument `object` given, returns `object`:

```
Integer(1)               # => 1
Integer(-1)              # => -1
```

With floating-point argument `object` given, returns `object` truncated to an integer:

```
Integer(1.9)             # => 1  # Rounds toward zero.
Integer(-1.9)            # => -1 # Rounds toward zero.
```

string objects

With string argument `object` and zero `base` given, returns `object` converted to an integer in base 10:

```
Integer('100')    # => 100
Integer('-100')   # => -100
```

With `base` zero, string `object` may contain leading characters to specify the actual base (radix indicator):

```
Integer('0100')  # => 64  # Leading '0' specifies base 8.
Integer('0b100') # => 4   # Leading '0b', specifies base 2.
Integer('0x100') # => 256 # Leading '0x' specifies base 16.
```

With a positive `base` (in range 2..36) given, returns `object` converted to an integer in the given base:

```
Integer('100', 2)   # => 4
Integer('100', 8)   # => 64
Integer('-100', 16) # => -256
```

With a negative `base` (in range -36..-2) given, returns `object` converted to an integer in the radix indicator if exists or `-base`:

```
Integer('0x100', -2)   # => 256
Integer('100', -2)     # => 4
Integer('0b100', -8)   # => 4
Integer('100', -8)     # => 64
Integer('0o100', -10)  # => 64
Integer('100', -10)    # => 100
```

`base` -1 is equal the -10 case.

When converting strings, surrounding whitespace and embedded underscores are allowed and ignored:

```
Integer(' 100 ')      # => 100
Integer('-1_0_0', 16) # => -256
```

other classes

Examples with `object` of various other classes:

```
Integer(Rational(9, 10)) # => 0  # Rounds toward zero.
Integer(Complex(2, 0))   # => 2  # Imaginary part must be zero.
Integer(Time.now)        # => 1650974042
```

keywords

With optional keyword argument `exception` given as `true` (the default):

- Raises `TypeError` if `object` does not respond to `to_int` or `to_i`.
- Raises `TypeError` if `object` is `nil`.
- Raise `ArgumentError` if `object` is an invalid string.

With `exception` given as `false`, an exception of any kind is suppressed and `nil` is returned.

Rational(x, y, exception: true) → rational or nil
Rational(arg, exception: true) → rational or nil

Returns `x/y` or `arg` as a Rational.

```
Rational(2, 3)   #=> (2/3)
Rational(5)      #=> (5/1)
Rational(0.5)    #=> (1/2)
Rational(0.3)    #=> (5404319552844595/18014398509481984)

Rational("2/3")  #=> (2/3)
Rational("0.3")  #=> (3/10)

Rational("10 cents")  #=> ArgumentError
Rational(nil)         #=> TypeError
Rational(1, nil)      #=> TypeError

Rational("10 cents", exception: false)  #=> nil
```

Syntax of the string form:

```
string form = extra spaces , rational , extra spaces ;
rational = [ sign ] , unsigned rational ;
unsigned rational = numerator | numerator , "/" , denominator ;
numerator = integer part | fractional part | integer part , fractional part ;
denominator = digits ;
integer part = digits ;
fractional part = "." , digits , [ ( "e" | "E" ) , [ sign ] , digits ] ;
sign = "-" | "+" ;
digits = digit , { digit | "_" , digit } ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
extra spaces = ? \s* ? ;
```

See also String#to_r.

String(object) → object or new_string

Returns a string converted from `object`.

Tries to convert `object` to a string using `to_str` first and `to_s` second:

```
String([0, 1, 2])         # => "[0, 1, 2]"
String(0..5)              # => "0..5"
String({foo: 0, bar: 1}) # => "{:foo=>0, :bar=>1}"
```

Raises `TypeError` if `object` cannot be converted to a string.

`__callee__ → symbol`

Returns the called name of the current method as a `Symbol`. If called outside of a method, it returns `nil`.

`__dir__ → string`

Returns the canonicalized absolute path of the directory of the file from which this method is called. It means symlinks in the path is resolved. If `__FILE__` is `nil`, it returns `nil`. The return value equals to `File.dirname(File.realpath(__FILE__))`.

`__method__ → symbol`

Returns the name at the definition of the current method as a `Symbol`. If called outside of a method, it returns `nil`.

`` `command` → string ``

Returns the `$stdout` output from running `command` in a subshell; sets global variable `$?` to the process status.

This method has potential security vulnerabilities if called with untrusted input; see Command Injection.

Examples:

```
$ `date`                 # => "Wed Apr  9 08:56:30 CDT 2003\n"
$ `echo oops && exit 99` # => "oops\n"
$ $?                     # => #<Process::Status: pid 17088 exit 99>
$ $?.status              # => 99>
```

The built-in syntax `%x{...}` uses this method.

`abort`
`abort(msg = nil)`

Terminates execution immediately, effectively by calling `Kernel.exit(false)`.

If string argument `msg` is given, it is written to STDERR prior to termination; otherwise, if an exception was raised, prints its message and backtrace.

`at_exit { block } → proc`

Converts *block* to a `Proc` object (and therefore binds it at the point of call) and registers it for execution when the program exits. If multiple handlers are registered,

they are executed in reverse order of registration.

```
def do_at_exit(str1)
  at_exit { print str1 }
end
at_exit { puts "cruel world" }
do_at_exit("goodbye ")
exit
```

produces:

```
goodbye cruel world
```

autoload(const, filename) → nil

```
Registers _filename_ to be loaded (using Kernel::require)
the first time that _const_ (which may be a String or
a symbol) is accessed.

   autoload(:MyModule, "/usr/local/lib/modules/my_module.rb")
```

If *const* is defined as autoload, the file name to be loaded is replaced with *filename*. If *const* is defined but not as autoload, does nothing.

autoload?(name, inherit=true) → String or nil

Returns *filename* to be loaded if *name* is registered as `autoload`.

```
autoload(:B, "b")
autoload?(:B)            #=> "b"
```

binding → a_binding

Returns a [Binding](#) object, describing the variable and method bindings at the point of call. This object can be used when calling [Binding#eval](#) to execute the evaluated command in this environment, or extracting its local variables.

```
class User
  def initialize(name, position)
    @name = name
    @position = position
  end

  def get_binding
    binding
  end
end
```

```
user = User.new('Joan', 'manager')
template = '{name: @name, position: @position}'

# evaluate template in context of the object
eval(template, user.get_binding)
#=> {:name=>"Joan", :position=>"manager"}
```

Binding#local_variable_get can be used to access the variables whose names are reserved Ruby keywords:

```
# This is valid parameter declaration, but `if` parameter can't
# be accessed by name, because it is a reserved word.
def validate(field, validation, if: nil)
  condition = binding.local_variable_get('if')
  return unless condition

  # ...Some implementation ...
end

validate(:name, :empty?, if: false) # skips validation
validate(:name, :empty?, if: true) # performs validation
```

block_given? → true or false

Returns `true` if `yield` would execute a block in the current context. The `iterator?` form is mildly deprecated.

```
def try
  if block_given?
    yield
  else
    "no block"
  end
end
try                  #=> "no block"
try { "hello" }      #=> "hello"
try do "hello" end   #=> "hello"
```

callcc {|cont| block } → obj

Generates a Continuation object, which it passes to the associated block. You need to `require 'continuation'` before using this method. Performing a *cont* `.call` will cause the callcc to return (as will falling through the end of the block). The value returned by the callcc is the value of the block, or the value passed to *cont* `.call`. See class Continuation for more details. Also see Kernel#throw for an alternative mechanism for unwinding a call stack.

caller(start=1, length=nil) → array or nil

caller(range) → array or nil

Returns the current execution stack—an array containing strings in the form `file:line` or `file:line: in `method'`.

The optional *start* parameter determines the number of initial stack entries to omit from the top of the stack.

A second optional `length` parameter can be used to limit how many entries are returned from the stack.

Returns `nil` if *start* is greater than the size of current execution stack.

Optionally you can pass a range, which will return an array containing the entries within the specified range.

```
def a(skip)
  caller(skip)
end
def b(skip)
  a(skip)
end
def c(skip)
  b(skip)
end
c(0)   #=> ["prog:2:in `a'", "prog:5:in `b'", "prog:8:in `c'", "prog:10:in `<
c(1)   #=> ["prog:5:in `b'", "prog:8:in `c'", "prog:11:in `<main>'"]
c(2)   #=> ["prog:8:in `c'", "prog:12:in `<main>'"]
c(3)   #=> ["prog:13:in `<main>'"]
c(4)   #=> []
c(5)   #=> nil
```

caller_locations(start=1, length=nil) → array or nil
caller_locations(range) → array or nil

Returns the current execution stack—an array containing backtrace location objects.

See Thread::Backtrace::Location for more information.

The optional *start* parameter determines the number of initial stack entries to omit from the top of the stack.

A second optional `length` parameter can be used to limit how many entries are returned from the stack.

Returns `nil` if *start* is greater than the size of current execution stack.

Optionally you can pass a range, which will return an array containing the entries within the specified range.

catch([tag]) {|tag| block } → obj

`catch` executes its block. If `throw` is not called, the block executes normally, and `catch` returns the value of the last expression evaluated.

```
catch(1) { 123 }            # => 123
```

If `throw(tag2, val)` is called, Ruby searches up its stack for a `catch` block whose `tag` has the same `object_id` as *tag2*. When found, the block stops executing and returns *val* (or `nil` if no second argument was given to `throw`).

```
catch(1) { throw(1, 456) }  # => 456
catch(1) { throw(1) }       # => nil
```

When `tag` is passed as the first argument, `catch` yields it as the parameter of the block.

```
catch(1) {|x| x + 2 }       # => 3
```

When no `tag` is given, `catch` yields a new unique object (as from `Object.new`) as the block parameter. This object can then be used as the argument to `throw`, and will match the correct `catch` block.

```
catch do |obj_A|
  catch do |obj_B|
    throw(obj_B, 123)
    puts "This puts is not reached"
  end

  puts "This puts is displayed"
  456
end

# => 456

catch do |obj_A|
  catch do |obj_B|
    throw(obj_A, 123)
    puts "This puts is still not reached"
  end

  puts "Now this puts is also not reached"
  456
end

# => 123
```

chomp → $_
chomp(string) → $_

Equivalent to `$_ = $_.chomp(`*`string`*`)`. See String#chomp. Available only when -p/-n command line option specified.

chop → $_

Equivalent to `($_.dup).chop!`, except `nil` is never returned. See `String#chop!`. Available only when -p/-n command line option specified.

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      #=> Integer
self.class   #=> Object
```

clone(freeze: nil) → 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 value state of *obj*, unless the `:freeze` keyword argument is given with a false or true value. 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.

eval(string [, binding [, filename [,lineno]]]) → obj

Evaluates the Ruby expression(s) in *string*. If *binding* is given, which must be a `Binding` object, the evaluation is performed in its context. If the optional *filename* and *lineno* parameters are present, they will be used when reporting syntax errors.

```
def get_binding(str)
  return binding
end
str = "hello"
eval "str + ' Fred'"                      #=> "hello Fred"
eval "str + ' Fred'", get_binding("bye")  #=> "bye Fred"
```

exec([env,] command_line, options = {})

exec([env,] exe_path, *args, options = {})

Replaces the current process by doing one of the following:

- Passing string `command_line` to the shell.
- Invoking the executable at `exe_path`.

This method has potential security vulnerabilities if called with untrusted input; see Command Injection.

The new process is created using the exec system call; it may inherit some of its environment from the calling program (possibly including open file descriptors).

Argument `env`, if given, is a hash that affects `ENV` for the new process; see Execution Environment.

Argument `options` is a hash of options for the new process; see Execution Options.

The first required argument is one of the following:

- `command_line` if it is a string, and if it begins with a shell reserved word or special built-in, or if it contains one or more metacharacters.
- `exe_path` otherwise.

Argument command_line

String argument `command_line` is a command line to be passed to a shell; it must begin with a shell reserved word, begin with a special built-in, or contain meta characters:

```
exec('echo')                       # Built-in.
exec('if true; then echo "Foo"; fi') # Shell reserved word.
exec('date > date.tmp')            # Contains meta character.
```

The command line may also contain arguments and options for the command:

```
exec('echo "Foo"')
```

Output:

```
Foo
```

On a Unix-like system, the shell is `/bin/sh`; otherwise the shell is determined by environment variable `ENV['RUBYSHELL']`, if defined, or `ENV['COMSPEC']` otherwise.

Except for the `COMSPEC` case, the entire string `command_line` is passed as an argument to shell option -c.

The shell performs normal shell expansion on the command line:

```
exec('echo C*')
```

Output:

```
CONTRIBUTING.md COPYING COPYING.ja
```

Raises an exception if the new process could not execute.

Argument `exe_path`

Argument `exe_path` is one of the following:

- The string path to an executable to be called.
- A 2-element array containing the path to an executable and the string to be used as the name of the executing process.

Example:

```
exec('/usr/bin/date')
```

Output:

```
Sat Aug 26 09:38:00 AM CDT 2023
```

Ruby invokes the executable directly, with no shell and no shell expansion:

```
exec('doesnt_exist') # Raises Errno::ENOENT
```

If one or more `args` is given, each is an argument or option to be passed to the executable:

```
exec('echo', 'C*')
exec('echo', 'hello', 'world')
```

Output:

```
C*
hello world
```

Raises an exception if the new process could not execute.

`exit(status = true)`

`exit(status = true)`

Initiates termination of the Ruby script by raising `SystemExit`; the exception may be caught. Returns exit status `status` to the underlying operating system.

Values `true` and `false` for argument `status` indicate, respectively, success and failure; The meanings of integer values are system-dependent.

Example:

```
begin
  exit
  puts 'Never get here.'
rescue SystemExit
  puts 'Rescued a SystemExit exception.'
end
puts 'After begin block.'
```

Output:

```
Rescued a SystemExit exception.
After begin block.
```

Just prior to final termination, Ruby executes any at-exit procedures (see Kernel::at_exit) and any object finalizers (see `ObjectSpace::define_finalizer`).

Example:

```
at_exit { puts 'In at_exit function.' }
ObjectSpace.define_finalizer('string', proc { puts 'In finalizer.' })
exit
```

Output:

```
In at_exit function.
In finalizer.
```

exit!(status = false)
exit!(status = false)

Exits the process immediately; no exit handlers are called. Returns exit status `status` to the underlying operating system.

```
Process.exit!(true)
```

Values `true` and `false` for argument `status` indicate, respectively, success and failure; The meanings of integer values are system-dependent.

fail

fail(string, cause: $!)
fail(exception [, string [, array]], cause: $!)

With no arguments, raises the exception in `$!` or raises a `RuntimeError` if `$!` is `nil`. With a single `String` argument, raises a `RuntimeError` with the string as a message. Otherwise, the first parameter should be an `Exception` class (or another object that returns an `Exception` object when sent an `exception` message). The optional second parameter sets the message associated with the exception (accessible via `Exception#message`), and the third parameter is an array of callback information (accessible via `Exception#backtrace`). The `cause` of the generated exception (accessible via `Exception#cause`) is automatically set to the "current" exception (`$!`), if any. An alternative value, either an `Exception` object or `nil`, can be specified via the `:cause` argument.

Exceptions are caught by the `rescue` clause of `begin...end` blocks.

```
raise "Failed to create socket"
raise ArgumentError, "No parameters", caller
```

Alias for: raise

fork { ... } → integer or nil
fork → integer or nil

Creates a child process.

With a block given, runs the block in the child process; on block exit, the child terminates with a status of zero:

```
puts "Before the fork: #{Process.pid}"
fork do
  puts "In the child process: #{Process.pid}"
end                   # => 382141
puts "After the fork: #{Process.pid}"
```

Output:

```
Before the fork: 420496
After the fork: 420496
In the child process: 420520
```

With no block given, the `fork` call returns twice:

- Once in the parent process, returning the pid of the child process.
- Once in the child process, returning `nil`.

Example:

```
puts "This is the first line before the fork (pid #{Process.pid})"
puts fork
puts "This is the second line after the fork (pid #{Process.pid})"
```

Output:

```
This is the first line before the fork (pid 420199)
420223
This is the second line after the fork (pid 420199)

This is the second line after the fork (pid 420223)
```

In either case, the child process may exit using `Kernel.exit!` to avoid the call to `Kernel#at_exit`.

To avoid zombie processes, the parent process should call either:

- `Process.wait`, to collect the termination statuses of its children.
- `Process.detach`, to register disinterest in their status.

The thread calling `fork` is the only thread in the created child process; `fork` doesn't copy other threads.

Note that method `fork` is available on some platforms, but not on others:

```
Process.respond_to?(:fork) # => true # Would be false on some.
```

If not, you may use ::spawn instead of `fork`.

format(*args)

Returns the string resulting from formatting `objects` into `format_string`.

For details on `format_string`, see Format Specifications.

Alias for: sprintf

frozen? → true or false

Returns the freeze status of *obj*.

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

gets(sep=$/ [, getline_args]) → string or nil
gets(limit [, getline_args]) → string or nil

gets(sep, limit [, getline_args]) → string or nil

Returns (and assigns to `$_`) the next line from the list of files in `ARGV` (or `$*`), or from standard input if no files are present on the command line. Returns `nil` at end of file. The optional argument specifies the record separator. The separator is included with the contents of each record. A separator of `nil` reads the entire contents, and a zero-length separator reads the input one paragraph at a time, where paragraphs are divided by two consecutive newlines. If the first argument is an integer, or optional second argument is given, the returning string would not be longer than the given value in bytes. If multiple filenames are present in `ARGV`, `gets(nil)` will read the contents one file at a time.

```
ARGV << "testfile"
print while gets
```

produces:

```
This is line one
This is line two
This is line three
And so on...
```

The style of programming using `$_` as an implicit parameter is gradually losing favor in the Ruby community.

global_variables → array

Returns an array of the names of global variables. This includes special regexp global variables such as `$~` and `$+`, but does not include the numbered regexp global variables (`$1`, `$2`, etc.).

```
global_variables.grep /std/   #=> [:$stdin, :$stdout, :$stderr]
```

gsub(pattern, replacement) → $_
gsub(pattern) {|...| block } → $_

Equivalent to `$_.gsub...`, except that `$_` will be updated if substitution occurs. Available only when -p/-n command line option specified.

iterator? → true or false

Deprecated. Use block_given? instead.

lambda { |...| block } → a_proc

Equivalent to Proc.new, except the resulting Proc objects check the number of parameters passed when called.

load(filename, wrap=false) → true

Loads and executes the Ruby program in the file *filename*.

If the filename is an absolute path (e.g. starts with '/'), the file will be loaded directly using the absolute path.

If the filename is an explicit relative path (e.g. starts with './' or '../'), the file will be loaded using the relative path from the current directory.

Otherwise, the file will be searched for in the library directories listed in `$LOAD_PATH` (`$:`). If the file is found in a directory, it will attempt to load the file relative to that directory. If the file is not found in any of the directories in `$LOAD_PATH`, the file will be loaded using the relative path from the current directory.

If the file doesn't exist when there is an attempt to load it, a LoadError will be raised.

If the optional *wrap* parameter is `true`, the loaded script will be executed under an anonymous module, protecting the calling program's global namespace. If the optional *wrap* parameter is a module, the loaded script will be executed under the given module. In no circumstance will any local variables in the loaded file be propagated to the loading environment.

local_variables → array

Returns the names of the current local variables.

```
fred = 1
for i in 1..10
   # ...
end
local_variables   #=> [:fred, :i]
```

loop { block }
loop → an_enumerator

Repeatedly executes the block.

If no block is given, an enumerator is returned instead.

```
loop do
  print "Input: "
  line = gets
  break if !line or line =~ /^q/i
```

```
  # ...
end
```

StopIteration raised in the block breaks the loop. In this case, loop returns the "result" value stored in the exception.

```
enum = Enumerator.new { |y|
  y << "one"
  y << "two"
  :ok
}

result = loop {
  puts enum.next
} #=> :ok
```

open(path, mode = 'r', perm = 0666, **opts) → io or nil
open(path, mode = 'r', perm = 0666, **opts) {|io| ... } → obj

Creates an IO object connected to the given file.

This method has potential security vulnerabilities if called with untrusted input; see Command Injection.

With no block given, file stream is returned:

```
open('t.txt') # => #<File:t.txt>
```

With a block given, calls the block with the open file stream, then closes the stream:

```
open('t.txt') {|f| p f } # => #<File:t.txt (closed)>
```

Output:

```
#<File:t.txt>
```

See File.open for details.

p(object) → obj
p(*objects) → array of objects
p → nil

For each object `obj`, executes:

```
$stdout.write(obj.inspect, "\n")
```

With one object given, returns the object; with multiple objects given, returns an array containing the objects; with no object given, returns `nil`.

Examples:

```
r = Range.new(0, 4)
p r                  # => 0..4
p [r, r, r]          # => [0..4, 0..4, 0..4]
p                    # => nil
```

Output:

```
0..4
[0..4, 0..4, 0..4]
```

`Kernel#p` is designed for debugging purposes. Ruby implementations may define `Kernel#p` to be uninterruptible in whole or in part. On CRuby, `Kernel#p`'s writing of data is uninterruptible.

`print(*objects) → nil`

Equivalent to `$stdout.print(*objects)`, this method is the straightforward way to write to `$stdout`.

Writes the given objects to `$stdout`; returns `nil`. Appends the output record separator `$OUTPUT_RECORD_SEPARATOR` (`$\`), if it is not `nil`.

With argument `objects` given, for each object:

- Converts via its method `to_s` if not a string.
- Writes to `stdout`.
- If not the last object, writes the output field separator `$OUTPUT_FIELD_SEPARATOR` (`$,`) if it is not `nil`.

With default separators:

```
objects = [0, 0.0, Rational(0, 1), Complex(0, 0), :zero, 'zero']
$OUTPUT_RECORD_SEPARATOR
$OUTPUT_FIELD_SEPARATOR
print(*objects)
```

Output:

```
nil
nil
00.00/10+0izerozero
```

With specified separators:

```
$OUTPUT_RECORD_SEPARATOR = "\n"
$OUTPUT_FIELD_SEPARATOR = ','
print(*objects)
```

Output:

```
0,0.0,0/1,0+0i,zero,zero
```

With no argument given, writes the content of `$_` (which is usually the most recent user input):

```
gets  # Sets $_ to the most recent user input.
print # Prints $_.
```

`printf(format_string, *objects) → nil`
`printf(io, format_string, *objects) → nil`

Equivalent to:

```
io.write(sprintf(format_string, *objects))
```

For details on `format_string`, see Format Specifications.

With the single argument `format_string`, formats `objects` into the string, then writes the formatted string to $stdout:

```
printf('%4.4d %10s %2.2f', 24, 24, 24.0)
```

Output (on $stdout):

```
0024         24 24.00#
```

With arguments `io` and `format_string`, formats `objects` into the string, then writes the formatted string to `io`:

```
printf($stderr, '%4.4d %10s %2.2f', 24, 24, 24.0)
```

Output (on $stderr):

```
0024         24 24.00# => nil
```

With no arguments, does nothing.

proc { |...| block } → a_proc

Equivalent to Proc.new.

putc(int) → int

Equivalent to:

```
$stdout.putc(int)
```

See IO#putc for important information regarding multi-byte characters.

puts(*objects) → nil

Equivalent to

```
$stdout.puts(objects)
```

raise
raise(string, cause: $!)
raise(exception [, string [, array]], cause: $!)

With no arguments, raises the exception in `$!` or raises a RuntimeError if `$!` is `nil`. With a single `String` argument, raises a `RuntimeError` with the string as a message. Otherwise, the first parameter should be an `Exception` class (or another object that returns an `Exception` object when sent an `exception` message). The optional second parameter sets the message associated with the exception (accessible via Exception#message), and the third parameter is an array of callback information (accessible via Exception#backtrace). The `cause` of the generated exception (accessible via Exception#cause) is automatically set to the "current" exception (`$!`), if any. An alternative value, either an `Exception` object or `nil`, can be specified via the `:cause` argument.

Exceptions are caught by the `rescue` clause of `begin...end` blocks.

```
raise "Failed to create socket"
raise ArgumentError, "No parameters", caller
```

Also aliased as: fail

rand(max=0) → number

If called without an argument, or if `max.to_i.abs == 0`, rand returns a pseudo-random floating point number between 0.0 and 1.0, including 0.0 and excluding 1.0.

```
rand        #=> 0.2725926052826416
```

When `max.abs` is greater than or equal to 1, `rand` returns a pseudo-random integer greater than or equal to 0 and less than `max.to_i.abs`.

```
rand(100)   #=> 12
```

When `max` is a `Range`, `rand` returns a random number where `range.member?(number) == true`.

Negative or floating point values for `max` are allowed, but may give surprising results.

```
rand(-100) # => 87
rand(-0.5) # => 0.8130921818028143
rand(1.9)  # equivalent to rand(1), which is always 0
```

`Kernel.srand` may be used to ensure that sequences of random numbers are reproducible between different runs of a program.

See also `Random.rand`.

readline(sep = $/, chomp: false) → string
readline(limit, chomp: false) → string
readline(sep, limit, chomp: false) → string

Equivalent to method `Kernel#gets`, except that it raises an exception if called at end-of-stream:

```
$ cat t.txt | ruby -e "p readlines; readline"
["First line\n", "Second line\n", "\n", "Fourth line\n", "Fifth line\n"]
in `readline': end of file reached (EOFError)
```

Optional keyword argument `chomp` specifies whether line separators are to be omitted.

readlines(sep = $/, chomp: false, **enc_opts) → array
readlines(limit, chomp: false, **enc_opts) → array
readlines(sep, limit, chomp: false, **enc_opts) → array

Returns an array containing the lines returned by calling `Kernel#gets` until the end-of-stream is reached; (see Line IO).

With only string argument `sep` given, returns the remaining lines as determined by line separator `sep`, or `nil` if none; see Line Separator:

```
# Default separator.
$ cat t.txt | ruby -e "p readlines"
["First line\n", "Second line\n", "\n", "Fourth line\n", "Fifth line\n"]

# Specified separator.
$ cat t.txt | ruby -e "p readlines 'li'"
["First li", "ne\nSecond li", "ne\n\nFourth li", "ne\nFifth li", "ne\n"]

# Get-all separator.
$ cat t.txt | ruby -e "p readlines nil"
["First line\nSecond line\n\nFourth line\nFifth line\n"]

# Get-paragraph separator.
$ cat t.txt | ruby -e "p readlines ''"
["First line\nSecond line\n\n", "Fourth line\nFifth line\n"]
```

With only integer argument `limit` given, limits the number of bytes in the line; see Line Limit:

```
$cat t.txt | ruby -e "p readlines 10"
["First line", "\n", "Second lin", "e\n", "\n", "Fourth lin", "e\n", "Fifth l

$cat t.txt | ruby -e "p readlines 11"
["First line\n", "Second line", "\n", "\n", "Fourth line", "\n", "Fifth line\

$cat t.txt | ruby -e "p readlines 12"
["First line\n", "Second line\n", "\n", "Fourth line\n", "Fifth line\n"]
```

With arguments `sep` and `limit` given, combines the two behaviors; see Line Separator and Line Limit.

Optional keyword argument `chomp` specifies whether line separators are to be omitted:

```
$ cat t.txt | ruby -e "p readlines(chomp: true)"
["First line", "Second line", "", "Fourth line", "Fifth line"]
```

Optional keyword arguments `enc_opts` specify encoding options; see Encoding options.

`require(name) → true or false`

Loads the given `name`, returning `true` if successful and `false` if the feature is already loaded.

If the filename neither resolves to an absolute path nor starts with ‘./’ or ‘../’, the file will be searched for in the library directories listed in `$LOAD_PATH` (`$:`). If the filename starts with ‘./’ or ‘../’, resolution is based on `Dir.pwd`.

If the filename has the extension “.rb”, it is loaded as a source file; if the extension is “.so”, “.o”, or “.dll”, or the default shared library extension on the current platform, Ruby loads the shared library as a Ruby extension. Otherwise, Ruby tries adding “.rb”,

".so", and so on to the name until found. If the file named cannot be found, a `LoadError` will be raised.

For Ruby extensions the filename given may use any shared library extension. For example, on Linux the socket extension is "socket.so" and `require 'socket.dll'` will load the socket extension.

The absolute path of the loaded file is added to `$LOADED_FEATURES` (`$"`). A file will not be loaded again if its path already appears in `$"`. For example, `require 'a'; require './a'` will not load `a.rb` again.

```
require "my-library.rb"
require "db-driver"
```

Any constants or globals within the loaded source file will be available in the calling program's global namespace. However, local variables will not be propagated to the loading environment.

require_relative(string) → true or false

Ruby tries to load the library named *string* relative to the directory containing the requiring file. If the file does not exist a `LoadError` is raised. Returns `true` if the file was loaded and `false` if the file was already loaded before.

select(read_ios, write_ios = [], error_ios = [], timeout = nil) → array or nil

Invokes system call select(2), which monitors multiple file descriptors, waiting until one or more of the file descriptors becomes ready for some class of I/O operation.

Not implemented on all platforms.

Each of the arguments `read_ios`, `write_ios`, and `error_ios` is an array of `IO` objects.

Argument `timeout` is an integer timeout interval in seconds.

The method monitors the IO objects given in all three arrays, waiting for some to be ready; returns a 3-element array whose elements are:

- An array of the objects in `read_ios` that are ready for reading.
- An array of the objects in `write_ios` that are ready for writing.
- An array of the objects in `error_ios` have pending exceptions.

If no object becomes ready within the given `timeout`, `nil` is returned.

IO.select peeks the buffer of IO objects for testing readability. If the IO buffer is not empty, IO.select immediately notifies readability. This "peek" only happens for IO objects. It does not happen for IO-like objects such as OpenSSL::SSL::SSLSocket.

The best way to use IO.select is invoking it after non-blocking methods such as read_nonblock, write_nonblock, etc. The methods raise an exception which is extended by `IO::WaitReadable` or `IO::WaitWritable`. The modules notify how the caller should wait with IO.select. If `IO::WaitReadable` is raised, the caller should wait for reading. If `IO::WaitWritable` is raised, the caller should wait for writing.

So, blocking read (readpartial) can be emulated using read_nonblock and IO.select as follows:

```
begin
  result = io_like.read_nonblock(maxlen)
rescue IO::WaitReadable
  IO.select([io_like])
  retry
rescue IO::WaitWritable
  IO.select(nil, [io_like])
  retry
end
```

Especially, the combination of non-blocking methods and IO.select is preferred for `IO` like objects such as OpenSSL::SSL::SSLSocket. It has to_io method to return underlying `IO` object. `IO.select` calls to_io to obtain the file descriptor to wait.

This means that readability notified by IO.select doesn't mean readability from OpenSSL::SSL::SSLSocket object.

The most likely situation is that OpenSSL::SSL::SSLSocket buffers some data. IO.select doesn't see the buffer. So IO.select can block when OpenSSL::SSL::SSLSocket#readpartial doesn't block.

However, several more complicated situations exist.

SSL is a protocol which is sequence of records. The record consists of multiple bytes. So, the remote side of SSL sends a partial record, `IO.select` notifies readability but OpenSSL::SSL::SSLSocket cannot decrypt a byte and OpenSSL::SSL::SSLSocket#readpartial will block.

Also, the remote side can request SSL renegotiation which forces the local SSL engine to write some data. This means OpenSSL::SSL::SSLSocket#readpartial may invoke write system call and it can block. In such a situation, OpenSSL::SSL::SSLSocket#read_nonblock raises `IO::WaitWritable` instead of blocking. So, the caller should wait for ready for writability as above example.

The combination of non-blocking methods and IO.select is also useful for streams such as tty, pipe socket socket when multiple processes read from a stream.

Finally, Linux kernel developers don't guarantee that readability of select(2) means readability of following read(2) even for a single process; see select(2)

Invoking IO.select before `IO#readpartial` works well as usual. However it is not the best way to use IO.select.

The writability notified by select(2) doesn't show how many bytes are writable. `IO#write` method blocks until given whole string is written. So, `IO#write(two or`

`more bytes)` can block after writability is notified by IO.select. `IO#write_nonblock` is required to avoid the blocking.

Blocking write (write) can be emulated using write_nonblock and `IO.select` as follows: `IO::WaitReadable` should also be rescued for SSL renegotiation in OpenSSL::SSL::SSLSocket.

```
while 0 < string.bytesize
  begin
    written = io_like.write_nonblock(string)
  rescue IO::WaitReadable
    IO.select([io_like])
    retry
  rescue IO::WaitWritable
    IO.select(nil, [io_like])
    retry
  end
  string = string.byteslice(written..-1)
end
```

Example:

```
rp, wp = IO.pipe
mesg = "ping "
100.times {
  # IO.select follows IO#read.  Not the best way to use IO.select.
  rs, ws, = IO.select([rp], [wp])
  if r = rs[0]
    ret = r.read(5)
    print ret
    case ret
    when /ping/
      mesg = "pong\n"
    when /pong/
      mesg = "ping "
    end
  end
  if w = ws[0]
    w.write(mesg)
  end
}
```

Output:

```
ping pong
ping pong
ping pong
(snipped)
ping
```

set_trace_func(proc) → proc
set_trace_func(nil) → nil

Establishes *proc* as the handler for tracing, or disables tracing if the parameter is `nil`.

Note: this method is obsolete, please use `TracePoint` instead.

proc takes up to six parameters:

- an event name string
- a filename string
- a line number
- a method name symbol, or nil
- a binding, or nil
- the class, module, or nil

proc is invoked whenever an event occurs.

Events are:

`"c-call"` call a C-language routine

`"c-return"` return from a C-language routine

`"call"` call a Ruby method

`"class"` start a class or module definition

`"end"` finish a class or module definition

`"line"` execute code on a new line

`"raise"` raise an exception

`"return"` return from a Ruby method

Tracing is disabled within the context of *proc*.

```
class Test
  def test
    a = 1
    b = 2
  end
end

set_trace_func proc { |event, file, line, id, binding, class_or_module|
  printf "%8s %s:%-2d %16p %14p\n", event, file, line, id, class_or_module
}
t = Test.new
t.test
```

Produces:

```
c-return prog.rb:8   :set_trace_func         Kernel
    line prog.rb:11              nil            nil
  c-call prog.rb:11             :new          Class
  c-call prog.rb:11      :initialize    BasicObject
c-return prog.rb:11      :initialize    BasicObject
```

```
c-return prog.rb:11             :new          Class
    line prog.rb:12              nil            nil
    call prog.rb:2             :test           Test
    line prog.rb:3             :test           Test
    line prog.rb:4             :test           Test
  return prog.rb:5             :test           Test
```

sleep(secs = nil) → slept_secs

Suspends execution of the current thread for the number of seconds specified by numeric argument `secs`, or forever if `secs` is `nil`; returns the integer number of seconds suspended (rounded).

```
Time.new  # => 2008-03-08 19:56:19 +0900
sleep 1.2 # => 1
Time.new  # => 2008-03-08 19:56:20 +0900
sleep 1.9 # => 2
Time.new  # => 2008-03-08 19:56:22 +0900
```

spawn([env,] command_line, options = {}) → pid
spawn([env,] exe_path, *args, options = {}) → pid

Creates a new child process by doing one of the following in that process:

- Passing string `command_line` to the shell.
- Invoking the executable at `exe_path`.

This method has potential security vulnerabilities if called with untrusted input; see Command Injection.

Returns the process ID (pid) of the new process, without waiting for it to complete.

To avoid zombie processes, the parent process should call either:

- `Process.wait`, to collect the termination statuses of its children.
- `Process.detach`, to register disinterest in their status.

The new process is created using the exec system call; it may inherit some of its environment from the calling program (possibly including open file descriptors).

Argument `env`, if given, is a hash that affects `ENV` for the new process; see Execution Environment.

Argument `options` is a hash of options for the new process; see Execution Options.

The first required argument is one of the following:

- `command_line` if it is a string, and if it begins with a shell reserved word or special built-in, or if it contains one or more metacharacters.

- `exe_path` otherwise.

Argument `command_line`

String argument `command_line` is a command line to be passed to a shell; it must begin with a shell reserved word, begin with a special built-in, or contain meta characters:

```
spawn('echo')                          # => 798847
Process.wait                           # => 798847
spawn('if true; then echo "Foo"; fi') # => 798848
Process.wait                           # => 798848
spawn('date > /tmp/date.tmp')          # => 798879
Process.wait                           # => 798849
spawn('date > /nop/date.tmp')          # => 798882 # Issues error message.
Process.wait                           # => 798882
```

The command line may also contain arguments and options for the command:

```
spawn('echo "Foo"') # => 799031
Process.wait        # => 799031
```

Output:

```
Foo
```

On a Unix-like system, the shell is `/bin/sh`; otherwise the shell is determined by environment variable `ENV['RUBYSHELL']`, if defined, or `ENV['COMSPEC']` otherwise.

Except for the `COMSPEC` case, the entire string `command_line` is passed as an argument to shell option -c.

The shell performs normal shell expansion on the command line:

```
spawn('echo C*') # => 799139
Process.wait     # => 799139
```

Output:

```
CONTRIBUTING.md COPYING COPYING.ja
```

Raises an exception if the new process could not execute.

Argument `exe_path`

Argument `exe_path` is one of the following:

- The string path to an executable to be called.
- A 2-element array containing the path to an executable and the string to be used as the name of the executing process.

Example:

```
spawn('/usr/bin/date') # => 799198 # Path to date on Unix-style system.
Process.wait           # => 799198
```

Output:

```
Thu Aug 31 10:06:48 AM CDT 2023
```

Ruby invokes the executable directly, with no shell and no shell expansion.

If one or more `args` is given, each is an argument or option to be passed to the executable:

```
spawn('echo', 'C*')             # => 799392
Process.wait                    # => 799392
spawn('echo', 'hello', 'world') # => 799393
Process.wait                    # => 799393
```

Output:

```
C*
hello world
```

Raises an exception if the new process could not execute.

sprintf(format_string *objects) → string

Returns the string resulting from formatting `objects` into `format_string`.

For details on `format_string`, see Format Specifications.

Also aliased as: format

srand(number = Random.new_seed) → old_seed

Seeds the system pseudo-random number generator, with `number`. The previous seed value is returned.

If `number` is omitted, seeds the generator using a source of entropy provided by the operating system, if available (/dev/urandom on Unix systems or the RSA cryptographic provider on Windows), which is then combined with the time, the process id, and a sequence number.

srand may be used to ensure repeatable sequences of pseudo-random numbers between different runs of the program. By setting the seed to a known value, programs can be made deterministic during testing.

```
srand 1234                 # => 268519324636777531569100071560086917274
[ rand, rand ]             # => [0.1915194503788923, 0.6221087710398319]
[ rand(10), rand(1000) ] # => [4, 664]
srand 1234                 # => 1234
[ rand, rand ]             # => [0.1915194503788923, 0.6221087710398319]
```

sub(pattern, replacement) → $_
sub(pattern) {|...| block } → $_

Equivalent to `$_.sub(args)`, except that `$_` will be updated if substitution occurs. Available only when -p/-n command line option specified.

syscall(integer_callno, *arguments) → integer

Invokes Posix system call syscall(2), which calls a specified function.

Calls the operating system function identified by `integer_callno`; returns the result of the function or raises `SystemCallError` if it failed. The effect of the call is platform-dependent. The arguments and returned value are platform-dependent.

For each of `arguments`: if it is an integer, it is passed directly; if it is a string, it is interpreted as a binary sequence of bytes. There may be as many as nine such arguments.

Arguments `integer_callno` and `argument`, as well as the returned value, are platform-dependent.

Note: `Method` `syscall` is essentially unsafe and unportable. The DL (Fiddle) library is preferred for safer and a bit more portable programming.

Not implemented on all platforms.

system([env,] command_line, options = {}, exception: false) → true, false, or nil
system([env,] exe_path, *args, options = {}, exception: false) → true, false, or nil

Creates a new child process by doing one of the following in that process:

- Passing string `command_line` to the shell.
- Invoking the executable at `exe_path`.

This method has potential security vulnerabilities if called with untrusted input; see Command Injection.

Returns:

- `true` if the command exits with status zero.

- `false` if the exit status is a non-zero integer.
- `nil` if the command could not execute.

Raises an exception (instead of returning `false` or `nil`) if keyword argument `exception` is set to `true`.

Assigns the command's error status to `$?`.

The new process is created using the system system call; it may inherit some of its environment from the calling program (possibly including open file descriptors).

Argument `env`, if given, is a hash that affects `ENV` for the new process; see Execution Environment.

Argument `options` is a hash of options for the new process; see Execution Options.

The first required argument is one of the following:

- `command_line` if it is a string, and if it begins with a shell reserved word or special built-in, or if it contains one or more metacharacters.
- `exe_path` otherwise.

Argument `command_line`

String argument `command_line` is a command line to be passed to a shell; it must begin with a shell reserved word, begin with a special built-in, or contain meta characters:

```
system('echo')                                 # => true  # Built-in.
system('if true; then echo "Foo"; fi')         # => true  # Shell reserved w
system('date > /tmp/date.tmp')                 # => true  # Contains meta ch
system('date > /nop/date.tmp')                 # => false
system('date > /nop/date.tmp', exception: true) # Raises RuntimeError.
```

Assigns the command's error status to `$?`:

```
system('echo')                                 # => true  # Built-in.
$?                                             # => #<Process::Status: pid 640610
system('date > /nop/date.tmp')                 # => false
$?                                             # => #<Process::Status: pid 640742
```

The command line may also contain arguments and options for the command:

```
system('echo "Foo"') # => true
```

Output:

```
Foo
```

On a Unix-like system, the shell is `/bin/sh`; otherwise the shell is determined by environment variable `ENV['RUBYSHELL']`, if defined, or `ENV['COMSPEC']` otherwise.

Except for the `COMSPEC` case, the entire string `command_line` is passed as an argument to shell option -c.

The shell performs normal shell expansion on the command line:

```
system('echo C*') # => true
```

Output:

```
CONTRIBUTING.md COPYING COPYING.ja
```

Raises an exception if the new process could not execute.

Argument `exe_path`

Argument `exe_path` is one of the following:

- The string path to an executable to be called.
- A 2-element array containing the path to an executable and the string to be used as the name of the executing process.

Example:

```
system('/usr/bin/date') # => true # Path to date on Unix-style system.
system('foo')           # => nil  # Command failed.
```

Output:

```
Mon Aug 28 11:43:10 AM CDT 2023
```

Assigns the command's error status to `$?`:

```
system('/usr/bin/date') # => true
$?                      # => #<Process::Status: pid 645605 exit 0>
system('foo')           # => nil
$?                      # => #<Process::Status: pid 645608 exit 127>
```

Ruby invokes the executable directly, with no shell and no shell expansion:

```
system('doesnt_exist') # => nil
```

If one or more `args` is given, each is an argument or option to be passed to the executable:

```
system('echo', 'C*')             # => true
system('echo', 'hello', 'world') # => true
```

Output:

```
C*
hello world
```

Raises an exception if the new process could not execute.

tap {|x| block } → 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}" }
  .to_a                  .tap {|x| puts "array:    #{x}" }
  .select {|x| x.even? } .tap {|x| puts "evens:    #{x}" }
  .map {|x| x*x }        .tap {|x| puts "squares:  #{x}" }
```

test(cmd, file1 [, file2]) → obj

Uses the character `cmd` to perform various tests on `file1` (first table below) or on `file1` and `file2` (second table).

File tests on a single file:

```
Cmd    Returns   Meaning
"A"  | Time    | Last access time for file1
"b"  | boolean | True if file1 is a block device
"c"  | boolean | True if file1 is a character device
"C"  | Time    | Last change time for file1
"d"  | boolean | True if file1 exists and is a directory
"e"  | boolean | True if file1 exists
"f"  | boolean | True if file1 exists and is a regular file
"g"  | boolean | True if file1 has the setgid bit set
"G"  | boolean | True if file1 exists and has a group
     |         | ownership equal to the caller's group
"k"  | boolean | True if file1 exists and has the sticky bit set
"l"  | boolean | True if file1 exists and is a symbolic link
"M"  | Time    | Last modification time for file1
"o"  | boolean | True if file1 exists and is owned by
     |         | the caller's effective uid
"O"  | boolean | True if file1 exists and is owned by
     |         | the caller's real uid
"p"  | boolean | True if file1 exists and is a fifo
"r"  | boolean | True if file1 is readable by the effective
     |         | uid/gid of the caller
"R"  | boolean | True if file is readable by the real
     |         | uid/gid of the caller
"s"  | int/nil | If file1 has nonzero size, return the size,
```

```
     |         | otherwise return nil
"S"  | boolean | True if file1 exists and is a socket
"u"  | boolean | True if file1 has the setuid bit set
"w"  | boolean | True if file1 exists and is writable by
     |         | the effective uid/gid
"W"  | boolean | True if file1 exists and is writable by
     |         | the real uid/gid
"x"  | boolean | True if file1 exists and is executable by
     |         | the effective uid/gid
"X"  | boolean | True if file1 exists and is executable by
     |         | the real uid/gid
"z"  | boolean | True if file1 exists and has a zero length
```

Tests that take two files:

```
"-"  | boolean | True if file1 and file2 are identical
"="  | boolean | True if the modification times of file1
     |         | and file2 are equal
"<"  | boolean | True if the modification time of file1
     |         | is prior to that of file2
">"  | boolean | True if the modification time of file1
     |         | is after that of file2
```

then {|x| block } → an_object

Yields self to the block and returns the result of the block.

```
3.next.then {|x| x**x }.to_s             #=> "256"
```

Good usage for `then` is value piping in method chains:

```
require 'open-uri'
require 'json'

construct_url(arguments).
  then {|url| URI(url).read }.
  then {|response| JSON.parse(response) }
```

When called without block, the method returns `Enumerator`, which can be used, for example, for conditional circuit-breaking:

```
# meets condition, no-op
1.then.detect(&:odd?)            # => 1
# does not meet condition, drop value
2.then.detect(&:odd?)            # => nil
```

throw(tag [, obj])

Transfers control to the end of the active `catch` block waiting for *tag*. Raises `UncaughtThrowError` if there is no `catch` block for the *tag*. The optional second parameter supplies a return value for the `catch` block, which otherwise defaults to `nil`. For examples, see Kernel::catch.

trace_var(symbol, cmd) → nil
trace_var(symbol) {|val| block } → nil

Controls tracing of assignments to global variables. The parameter `symbol` identifies the variable (as either a string name or a symbol identifier). *cmd* (which may be a string or a `Proc` object) or block is executed whenever the variable is assigned. The block or `Proc` object receives the variable's new value as a parameter. Also see Kernel::untrace_var.

```
trace_var :$_, proc {|v| puts "$_ is now '#{v}'" }
$_ = "hello"
$_ = ' there'
```

produces:

```
$_ is now 'hello'
$_ is now ' there'
```

trap(signal, command) → obj
trap(signal) {| | block } → obj

Specifies the handling of signals. The first parameter is a signal name (a string such as "SIGALRM", "SIGUSR1", and so on) or a signal number. The characters "SIG" may be omitted from the signal name. The command or block specifies code to be run when the signal is raised. If the command is the string "IGNORE" or "SIG_IGN", the signal will be ignored. If the command is "DEFAULT" or "SIG_DFL", the Ruby's default handler will be invoked. If the command is "EXIT", the script will be terminated by the signal. If the command is "SYSTEM_DEFAULT", the operating system's default handler will be invoked. Otherwise, the given command or block will be run. The special signal name "EXIT" or signal number zero will be invoked just prior to program termination. trap returns the previous handler for the given signal.

```
Signal.trap(0, proc { puts "Terminating: #{$$}" })
Signal.trap("CLD")  { puts "Child died" }
fork && Process.wait
```

produces:

```
Terminating: 27461
Child died
```

```
Terminating: 27460
```

untrace_var(symbol [, cmd]) → array or nil

Removes tracing for the specified command on the given global variable and returns `nil`. If no command is specified, removes all tracing for that variable and returns an array containing the commands actually removed.

warn(*msgs, uplevel: nil, category: nil) → nil

If warnings have been disabled (for example with the `-W0` flag), does nothing. Otherwise, converts each of the messages to strings, appends a newline character to the string if the string does not end in a newline, and calls `Warning.warn` with the string.

```
warn("warning 1", "warning 2")
```

produces:

```
warning 1
warning 2
```

If the `uplevel` keyword argument is given, the string will be prepended with information for the given caller frame in the same format used by the `rb_warn` C function.

```
# In baz.rb
def foo
  warn("invalid call to foo", uplevel: 1)
end

def bar
  foo
end

bar
```

produces:

```
baz.rb:6: warning: invalid call to foo
```

If `category` keyword argument is given, passes the category to `Warning.warn`. The category given must be be one of the following categories:

:deprecated Used for warning for deprecated functionality that may be removed in the future.

:experimental Used for experimental features that may change in future releases.

yield_self {|x| block } → an_object

Yields self to the block and returns the result of the block.

```
"my string".yield_self {|s| s.upcase }   #=> "MY STRING"
```

Good usage for `then` is value piping in method chains:

```
require 'open-uri'
require 'json'

construct_url(arguments).
  then {|url| URI(url).read }.
  then {|response| JSON.parse(response) }
```

Private Instance Methods

pp(*objs)

suppress redefinition warning

Also aliased as: pp